CHAPTER 7

Advanced Encryption Standard (AES)

Objectives

In this chapter, we discuss the Advanced Encryption Standard (AES), the modern symmetric-key block cipher that may replace DES. This chapter has several objectives:

- ❏ To review a short history of AES
- ❏ To define the basic structure of AES
- ❏ To define the transformations used by AES
- ❏ To define the key expansion process
- ❏ To discuss different implementations

The emphasis is on how the algebraic structures discussed in Chapter 4 achieve the AES security goals.

7.1 INTRODUCTION

The **Advanced Encryption Standard (AES)** is a symmetric-key block cipher published by the **National Institute of Standards and Technology** (**NIST**) in December 2001.

History

In 1997, NIST started looking for a replacement for DES, which would be called the *Advanced Encryption Standard* or *AES.* The NIST specifications required a block size of 128 bits and three different key sizes of 128, 192, and 256 bits. The specifications also required that AES be an open algorithm, available to the public worldwide. The announcement was made internationally to solicit responses from all over the world.

After the *First AES Candidate Conference,* NIST announced that 15 out of 21 received algorithms had met the requirements and been selected as the first candidates (August 1998). Algorithms were submitted from a number of countries; the

variety of these proposals demonstrated the openness of the process and worldwide participation.

After the *Second AES Candidate Conference,* which was held in Rome, NIST announced that 5 out of 15 candidates—*MARS*, *RC6*, *Rijndael*, *Serpent*, and *Twofish*—were selected as the finalists (August 1999).

After the *Third AES Candidate Conference,* NIST announced that **Rijndael,** (pronounced like "Rain Doll"), designed by Belgian researchers Joan Daemen and Vincent Rijment, was selected as *Advanced Encryption Standard* (October 2000).

In February 2001, NIST announced that a draft of the **Federal Information Processing Standard (FIPS)** was available for public review and comment.

Finally, AES was published as FIPS 197 in the *Federal Register* in December 2001.

Criteria

The criteria defined by NIST for selecting AES fall into three areas: security, cost, and implementation. At the end, *Rijndael* was judged the best at meeting the combination of these criteria.

Security

The main emphasis was on security. Because NIST explicitly demanded a 128-bit key, this criterion focused on resistance to cryptanalysis attacks other than brute-force attack.

Cost

The second criterion was cost, which covers the computational efficiency and storage requirement for different implementations such as hardware, software, or smart cards.

Implementation

This criterion included the requirement that the algorithm must have flexibility (be implementable on any platform) and simplicity.

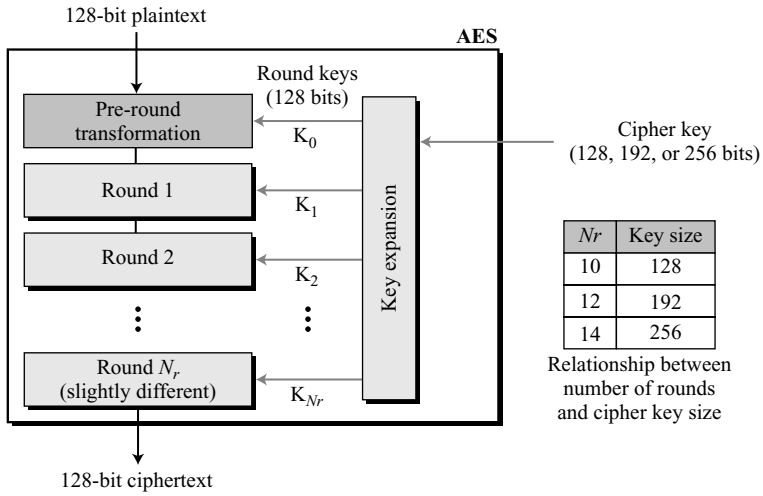
Rounds

AES is a non-Feistel cipher that encrypts and decrypts a data block of 128 bits. It uses 10, 12, or 14 rounds. The key size, which can be 128, 192, or 256 bits, depends on the number of rounds. Figure 7.1 shows the general design for the encryption algorithm (called cipher); the decryption algorithm (called inverse cipher) is similar, but the round keys are applied in the reverse order.

In Figure 7.1, N_r defines the number of rounds. The figure also shows the relationship between the number of rounds and the key size, which means that we can have three different AES versions; they are referred as AES-128, AES-192, and AES-256. However, the round keys, which are created by the key-expansion algorithm are always 128 bits, the same size as the plaintext or ciphertext block.

AES has defined three versions, with 10, 12, and 14 rounds.
Each version uses a different cipher key size (128, 192, or 256), but the round keys are always 128 bits.

Figure 7.1 *General design of AES encryption cipher*



Nr	Key size
10	128
12	192
14	256

Relationship between number of rounds and cipher key size

The number of round keys generated by the key-expansion algorithm is always one more than the number of rounds. In other words, we have

$$\textbf{Number of round keys} = N_r + 1$$

We refer to the round keys as K_0, K_1, K_2, …, K_{N_r}.

Data Units

AES uses five units of measurement to refer to data: bits, bytes, words, blocks, and state. The bit is the smallest and atomic unit; other units can be expressed in terms of smaller ones. Figure 7.2 shows the non-atomic data units: byte, word, block, and state.

Bit

In AES, a **bit** is a binary digit with a value of 0 or 1. We use a lowercase letter to refer to a bit.

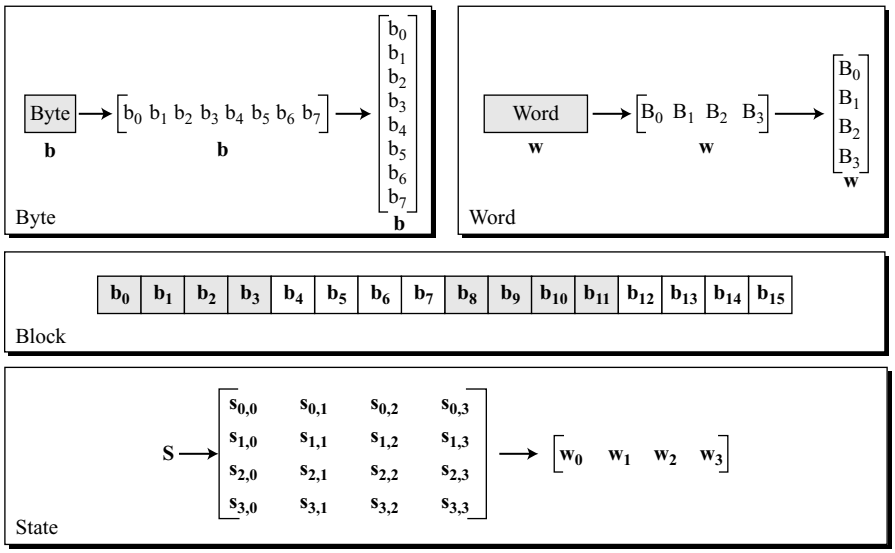
Byte

A **byte** is a group of eight bits that can be treated as a single entity, a row matrix (1×8) of eight bits, or a column matrix (8×1) of eight bits. When treated as a row matrix, the bits are inserted to the matrix from left to right; when treated as a column matrix, the bits are inserted into the matrix from top to bottom. We use a lowercase bold letter to refer to a byte.

Word

A **word** is a group of 32 bits that can be treated as a single entity, a row matrix of four bytes, or a column matrix of four bytes. When it is treated as a row matrix, the bytes are

Figure 7.2 *Data units used in AES*



inserted into the matrix from left to right; when it is considered as a column matrix, the bytes are inserted into the matrix from top to bottom. We use the lowercase bold letter **w** to show a word.

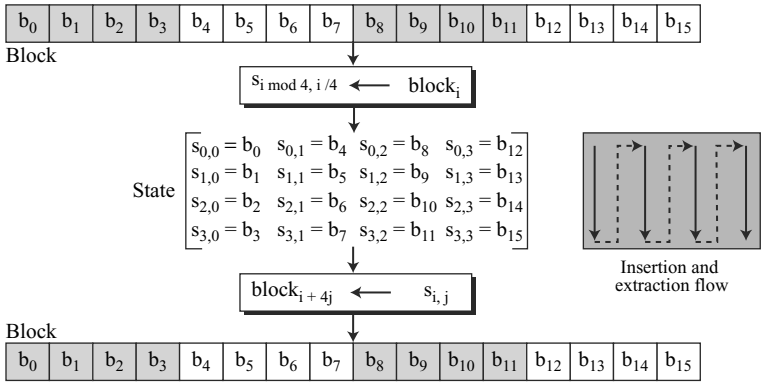
Block

AES encrypts and decrypts data blocks. A **block** in AES is a group of 128 bits. However, a block can be represented as a row matrix of 16 bytes.

State

AES uses several rounds in which each round is made of several stages. Data block is transformed from one stage to another. At the beginning and end of the cipher, AES uses the term *data block;* before and after each stage, the data block is referred to as a **state.** We use an uppercase bold letter to refer to a state. Although the states in different stages are normally called **S,** we occasionally use the letter **T** to refer to a temporary state. States, like blocks, are made of 16 bytes, but normally are treated as matrices of 4×4 bytes. In this case, each element of a state is referred to as $\mathbf{s}_{r,c}$, where r (0 to 3) defines the row and the c (0 to 3) defines the column. Occasionally, a state is treated as a row matrix (1×4) of words. This makes sense, if we think of a word as a column matrix. At the beginning of the cipher, bytes in a data block are inserted into a state column by column, and in each column, from top to bottom. At the end of the cipher, bytes in the state are extracted in the same way, as shown in Figure 7.3.

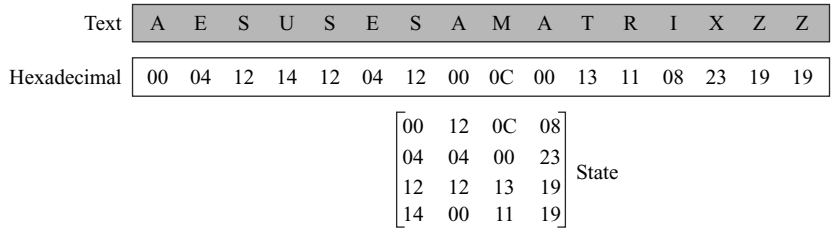
Figure 7.3 *Block-to-state and state-to-block transformation*



Example 7.1

Let us see how a 16-character block can be shown as a 4×4 matrix. Assume that the text block is "AES uses a matrix". We add two bogus characters at the end to get "AESUSESAMATRIXZZ". Now we replace each character with an integer between 00 and 25. We then show each byte as an integer with two hexadecimal digits. For example, the character "S" is first changed to 18 and then written as 12_{16} in hexadecimal. The state matrix is then filled up, column by column, as shown in Figure 7.4.

Figure 7.4 *Changing plaintext to state*

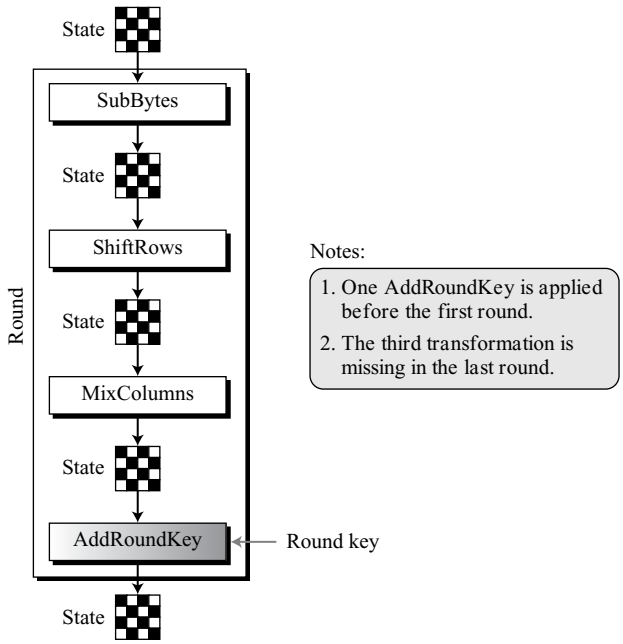


Structure of Each Round

Figure 7.5 shows the structure of each round at the encryption side. Each round, except the last, uses four transformations that are invertible. The last round has only three transformations.

As Figure 7.5 shows, each transformation takes a state and creates another state to be used for the next transformation or the next round. The pre-round section uses only one transformation (AddRoundKey); the last round uses only three transformations (MixColumns transformation is missing).

Figure 7.5 *Structure of each round at the encryption site*



At the decryption site, the inverse transformations are used: InvSubByte, InvShiftRows, InvMixColumns, and AddRoundKey (this one is self-invertible).

7.2 TRANSFORMATIONS

To provide security, AES uses four types of transformations: substitution, permutation, mixing, and key-adding. We will discuss each here.

Substitution

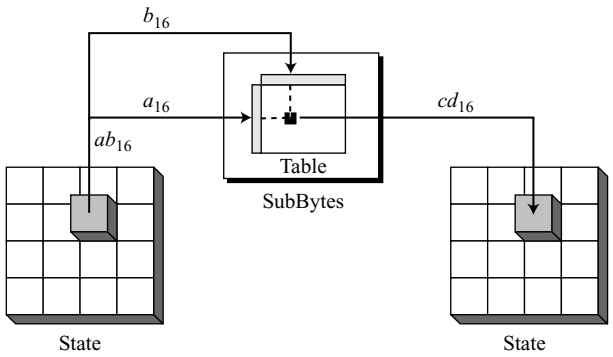
AES, like DES, uses substitution. However, the mechanism is different. First, the substitution is done for each byte. Second, only one table is used for transformation of every byte, which means that if two bytes are the same, the transformation is also the same. Third, the transformation is defined by either a table lookup process or mathematical calculation in the GF(2^8) field. AES uses two invertible transformations.

SubBytes

The first transformation, **SubBytes,** is used at the encryption site. To substitute a byte, we interpret the byte as two hexadecimal digits. The left digit defines the row

and the right digit defines the column of the substitution table. The two hexadecimal digits at the junction of the row and the column are the new byte. Figure 7.6 shows the idea.

Figure 7.6 *SubBytes transformations*



In the SubBytes transformation, the state is treated as a 4×4 matrix of bytes. Transformation is done one byte at a time. The contents of each byte is changed, but the arrangement of the bytes in the matrix remains the same. In the process, each byte is transformed independently. There are sixteen distinct byte-to-byte transformations.

The SubBytes operation involves 16 independent byte-to-byte transformations.

Table 7.1 shows the substitution table (S-box) for SubBytes transformation. The transformation definitely provides confusion effect. For example, two bytes, $5A_{16}$ and $5B_{16}$, which differ only in one bit (the rightmost bit) are transformed to BE_{16} and 39_{16}, which differ in four bits.

Table 7.1 *SubBytes transformation table*

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*A*	*B*	*C*	*D*	*E*	*F*
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8

Table 7.1 *SubBytes transformation table (continued)*

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

InvSubBytes

InvSubBytes is the inverse of SubBytes. The transformation is done using Table 7.2. We can easily check that the two transformations are inverse of each other.

Table 7.2 *InvSubBytes transformation table*

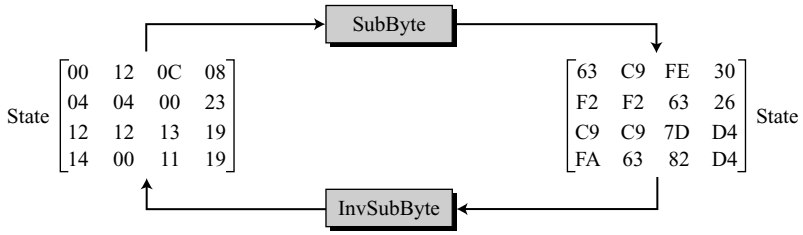
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*A*	*B*	*C*	*D*	*E*	*F*
0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
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

Example 7.2

Figure 7.7 shows how a state is transformed using the SubBytes transformation. The figure also shows that the InvSubBytes transformation creates the original one. Note that if the two bytes

have the same values, their transformation is also the same. For example, the two bytes 04_{16} and 04_{16} in the left state are transformed to $F2_{16}$ and $F2_{16}$ in the right state and vice versa.The reason is that every byte uses the same table. In contrast, we saw that DES (Chapter 6) uses eight different S-boxes.

Figure 7.7 *SubBytes transformation for Example 7.2*



Transformation Using the GF(2^8) Field

Although we can use Table 7.1 or Table 7.2 to find the substitution for each byte, AES also defines the transformation algebraically using the GF(2^8) field with the irreducible polynomials ($x^8 + x^4 + x^3 + x + 1$), as shown in Figure 7.8.

The SubBytes transformation repeats a routine, called *subbyte,* sixteen times. The InvSubBytes repeats a routine called *invsubbyte.* Each iteration transforms one byte.

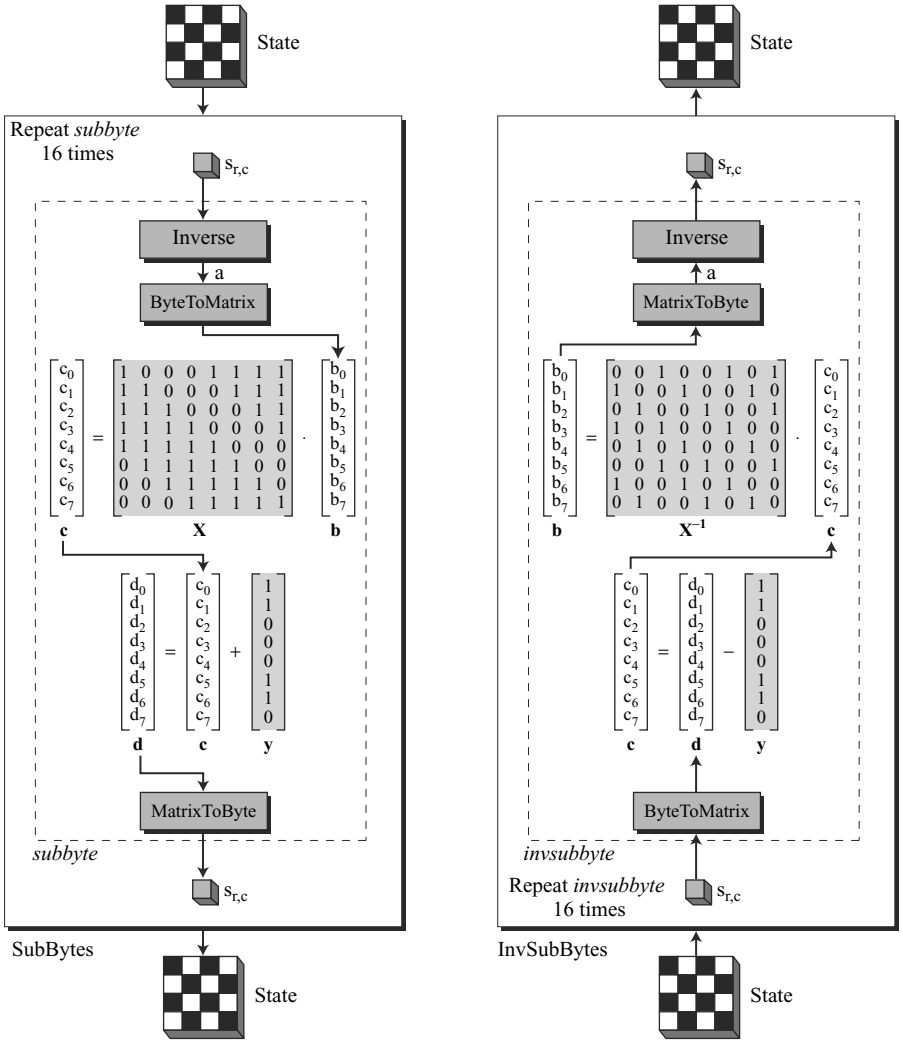
In the *subbyte* routine, the multiplicative inverse of the byte (as an 8-bit binary string) is found in GF(2^8) with the irreducible polynomial ($x^8 + x^4 + x^3 + x + 1$) as the modulus. Note that if the byte is 00_{16}, its inverse is itself. The inverted byte is then interpreted as a column matrix with the least significant bit at the top and the most significant bit at the bottom. This column matrix is multiplied by a constant square matrix, **X,** and the result, which is a column matrix, is added with a constant column matrix, **y,** to give the new byte. Note that multiplication and addition of bits are done in GF(2). The *invsubbyte* is doing the same thing in reverse order.

After finding the multiplicative inverse of the byte, the process is similar to the affine ciphers we discussed in Chapter 3. In the encryption, multiplication is first and addition is second; in the decryption, subtraction (addition by inverse) is first and division (multiplication by inverse) is second. We can easily prove that the two transformations are inverses of each other because addition or subtraction in GF(2) is actually the XOR operation.

$$\text{subbyte:} \quad \rightarrow \quad \mathbf{d} = \mathbf{X}\,(s_{r,c})^{-1} \oplus \mathbf{y}$$
$$\text{invsubbyte:} \quad \rightarrow \quad [\mathbf{X}^{-1}(\mathbf{d} \oplus \mathbf{y})]^{-1} = [\mathbf{X}^{-1}(\mathbf{X}\,(s_{r,c})^{-1} \oplus \mathbf{y} \oplus \mathbf{y})]^{-1} = [(s_{r,c})^{-1}]^{-1} = s_{r,c}$$

The SubBytes and InvSubBytes transformations are inverses of each other.

Figure 7.8 *SubBytes and InvSubBytes processes*



Example 7.3

Let us show how the byte 0C is transformed to FE by *subbyte* routine and transformed back to 0C by the *invsubbyte* routine.

1. *subbyte:*
   a. The multiplicative inverse of 0C in GF(2^8) field is B0, which means **b** is (10110000).
   b. Multiplying matrix **X** by this matrix results in **c** = (10011101)
   c. The result of XOR operation is **d** = (11111110), which is FE in hexadecimal.

2. *invsubbyte:*
   a. The result of XOR operation is **c** = (10011101)
   b. The result of multiplying by matrix $\mathbf{X}^{-1}$ is (11010000) or B0
   c. The multiplicative inverse of B0 is 0C.

Algorithm

Although we have shown matrices to emphasize the nature of substitution (affine transformation), the algorithm does not necessarily use multiplication and addition of matrices because most of the elements in the constant square matrix are only 0 or 1. The value of the constant column matrix is 0x63. We can write a simple algorithm to do the SubBytes. Algorithm 7.1 calls the subbyte routine 16 time, one for each byte in the state.

Algorithm 7.1 *Pseudocode for SubBytes transformation*

```
SubBytes (S)
{
   for (r = 0 to 3)
     for (c = 0 to 3)
           S_r,c = subbyte (S_r,c)
}

subbyte (byte)
{
   a ← byte^-1              // Multiplicative inverse in GF(2^8) with inverse of 00 to be 00
    ByteToMatrix (a, b)
    for (i = 0 to 7)
    {
        c_i ← b_i ⊕ b_(i+4)mod 8 ⊕ b_(i+5)mod 8 ⊕ b_(i+6)mod 8 ⊕ b_(i+7)mod 8
        d_i ← c_i ⊕ ByteToMatrix (0x63)
   }
    MatrixToByte (d, d)
    byte ← d
}
```

The ByteToMatrix routine transforms a byte to an 8×1 column matrix. The MatrixToByte routine transforms an 8×1 column matrix to a byte. The expansion of these routines and the algorithm for InvSubBytes are left as exercises.

Nonlinearity

Although the multiplication and addition of matrices in the *subbyte* routine are an affine-type transformation and linear, the replacement of the byte by its multiplicative inverse in GF(2^8) is nonlinear. This step makes the whole transformation nonlinear.

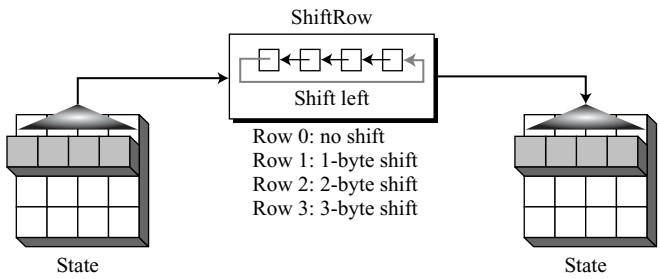
Permutation

Another transformation found in a round is shifting, which permutes the bytes. Unlike DES, in which permutation is done at the bit level, shifting transformation in AES is done at the byte level; the order of the bits in the byte is not changed.

ShiftRows

In the encryption, the transformation is called **ShiftRows** and the shifting is to the left.The number of shifts depends on the row number (0, 1, 2, or 3) of the state matrix. This means the row 0 is not shifted at all and the last row is shifted three bytes. Figure 7.9 shows the shifting transformation.

Figure 7.9 *ShiftRows transformation*



Note that the ShiftRows transformation operates one row at a time.

InvShiftRows

In the decryption, the transformation is called **InvShiftRows** and the shifting is to the right. The number of shifts is the same as the row number (0, 1, 2, and 3) of the state matrix.

The ShiftRows and InvShiftRows transformations are inverses of each other.

Algorithm

Algorithm 7.2 for ShiftRows transformation is very simple. However, to emphasize that the transformation is one row at a time, we use a routine called *shiftrow* that shifts the byte in a single row. We call this routine three times. The shiftrow routine first copies the row into a temporary row matrix, **t.** It then shifts the row.

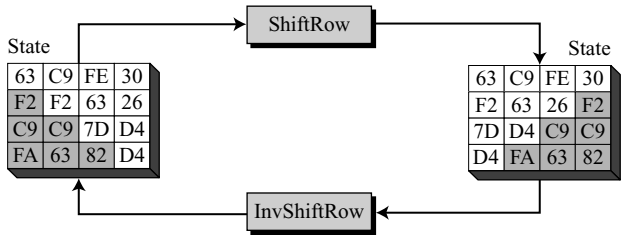
Example 7.4

Figure 7.10 shows how a state is transformed using ShiftRows transformation. The figure also shows that InvShiftRows transformation creates the original state.

Algorithm 7.2 *Pseudocode for ShiftRows transformation*

```
ShiftRows (S)
{
    for (r = 1 to 3)
        shiftrow (s_r, r)                    // s_r is the rth row
}

shiftrow (row, n)                            // n is the number of bytes to be shifted
{
   CopyRow (row, t)                          // t is a temporary row
   for (c = 0 to 3)
        row_((c – n) mod 4) ← t_c
}
```

Figure 7.10 *ShiftRows transformation in Example 7.4*

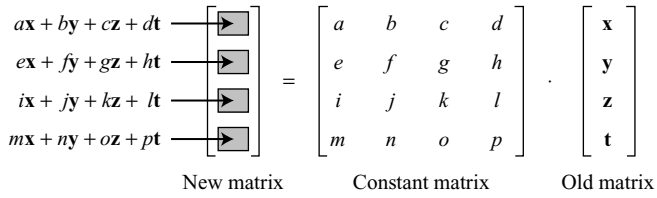


Mixing

The substitution provided by the SubBytes transformation changes the value of the byte based only on original value and an entry in the table; the process does not include the neighboring bytes. We can say that SubBytes is an *intrabyte* transformation. The permutation provided by the ShiftRows transformation exchanges bytes without permuting the bits inside the bytes. We can say that ShiftRows is a *byte-exchange* transformation. We also need an *interbyte* transformation that changes the bits inside a byte, based on the bits inside the neighboring bytes. We need to mix bytes to provide diffusion at the bit level.

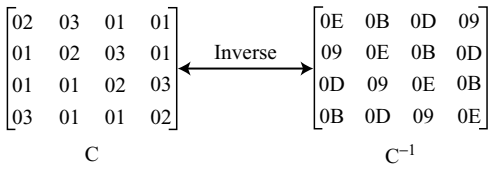
The mixing transformation changes the contents of each byte by taking four bytes at a time and combining them to recreate four new bytes. To guarantee that each new byte is different (even if all four bytes are the same), the combination process first multiplies each byte with a different constant and then mixes them. The mixing can be provided by matrix multiplication. As we discussed in Chapter 2, when we multiply a square matrix by a column matrix, the result is a new column matrix. Each element in the new matrix depends on all four elements of the old matrix after they are multiplied by row values in the constant matrix. Figure 7.11 shows the idea.

Figure 7.11 *Mixing bytes using matrix multiplication*



AES defines a transformation, called MixColumns, to achieve this goal. There is also an inverse transformation, called InvMixColumns. Figure 7.12 shows the constant matrices used for these transformations. These two matrices are inverses of each other when the elements are interpreted as 8-bit words (or polynomials) with coefficients in GF(2^8). The proof is left as an exercise.

Figure 7.12 *Constant matrices used by MixColumns and InvMixColumns*



MixColumns

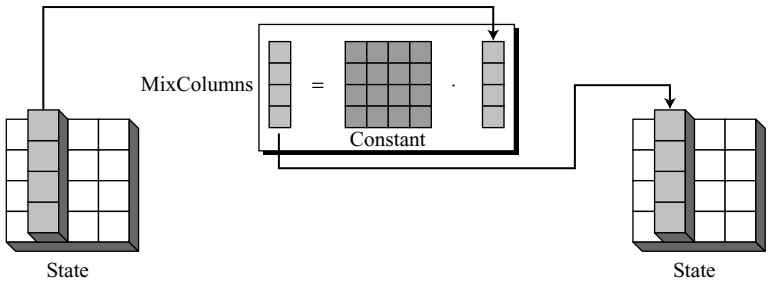
The **MixColumns** transformation operates at the column level; it transforms each column of the state to a new column. The transformation is actually the matrix multiplication of a state column by a constant square matrix. The bytes in the state column and constants matrix are interpreted as 8-bit words (or polynomials) with coefficients in GF(2). Multiplication of bytes is done in GF(2^8) with modulus (10001101) or ($x^8 + x^4 + x^3 + x + 1$). Addition is the same as XORing of 8-bit words. Figure 7.13 shows the MixColumns transformations.

InvMixColumns

The **InvMixColumns** transformation is basically the same as the MixColumns transformation. If the two constant matrices are inverses of each other, it is easy to prove that the two transformations are inverses of each other.

The MixColumns and InvMixColumns transformations are inverses of each other.

Figure 7.13 *MixColumns transformation*



Algorithm

Algorithm 7.3 shows the code for MixColumns transformation.

Algorithm 7.3 *Pseudocode for MixColumns transformation*

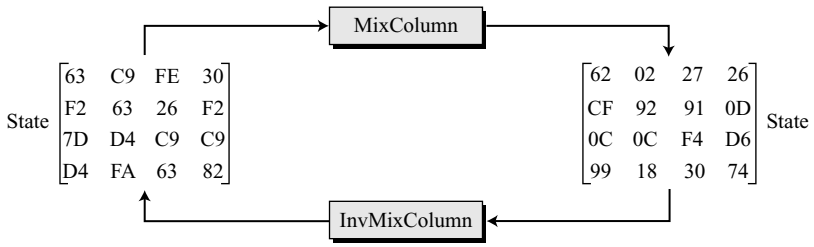
```
MixColumns (S)
{
    for (c = 0 to 3)
        mixcolumn (s_c)
}

mixcolumn (col)
{
   CopyColumn (col, t)                    // t is a temporary column

   col_0 ← (0x02) • t_0 ⊕ (0x03 • t_1) ⊕ t_2 ⊕ t_3

   col_1 ← t_0 ⊕ (0x02) • t_1 ⊕ (0x03) • t_2 ⊕ t_3

   col_2 ← t_0 ⊕ t_1 ⊕ (0x02) • t_2 ⊕ (0x03) • t_3

   col_3 ← (0x03 • t_0) ⊕ t_1 ⊕ t_2 ⊕ (0x02) • t_3
}
```

Algorithms for MixColumns and InvMixColumns involve multiplication and addition in the GF(2^8) field. As we saw in Chapter 4, there is a simple and efficient algorithm for multiplication and addition in this field. However, to show the nature of the algorithm (transformation of a column at a time), we use a routine, called *mixcolumn,* to be called four times by the algorithm. The routine *mixcolumn* simply multiplies the rows of the constant matrix by a column in the state. In the above algorithm, the operator (•) used in the mixcolumn routine is multiplication in the GF(2^8) field. It can be replaced with a simple routine as discussed in Chapter 4. The code for InvMixColumns is left as an exercise.

Example 7.5

Figure 7.14 shows how a state is transformed using the MixColumns transformation. The figure also shows that the InvMixColumns transformation creates the original one.

Figure 7.14 *The MixColumns transformation in Example 7.5*



Note that equal bytes in the old state are not equal any more in the new state. For example, the two bytes F2 in the second row are changed to CF and 0D.

Key Adding

Probably the most important transformation is the one that includes the cipher key. All previous transformations use known algorithms that are invertible. If the cipher key is not added to the state at each round, it is very easy for the adversary to find the plaintext, given the ciphertext. The cipher key is the only secret between Alice and Bob in this case.

AES uses a process called key expansion (discussed later in the Chapter) that creates N_r +1 round keys from the cipher key. Each round key is 128 bits long—it is treated as four 32-bit words. For the purpose of adding the key to the state, each word is considered as a column matrix.

AddRoundKey

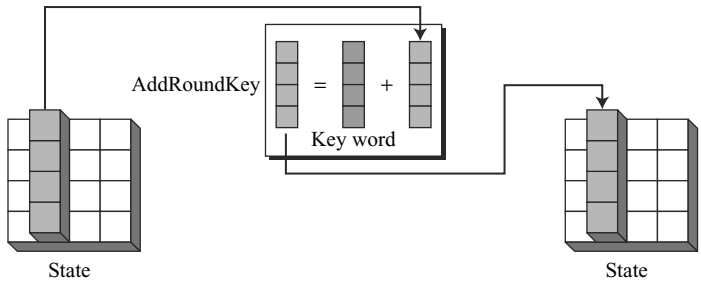
AddRoundKey also proceeds one column at a time. It is similar to MixColumns in this respect. MixColumns multiplies a constant square matrix by each state column; AddRoundKey adds a round key word with each state column matrix. The operation in MixColumns is matrix multiplication; the operation in AddRoundKey is matrix addition. Since addition and subtraction in this field are the same, the AddRoundKey transformation is the inverse of itself. Figure 7.15 shows the AddRoundKey transformation.

The AddRoundKey transformation is the inverse of itself.

Algorithm

The AddRoundKey transformation can be thought as XORing of each column of the state, with the corresponding key word. We will discuss how the cipher key is expanded

Figure 7.15 *AddRoundKey transformation*



into a set of key words, but for the moment we can define this transformation as shown in Algorithm 7.4. Note that $\mathbf{s}_c$ and $\mathbf{w}_{\text{round}+4c}$ are 4×1 column matrices.

Algorithm 7.4 *Pseudocode for AddRoundKey transformation*

```
AddRoundKey (S)
{
    for (c = 0 to 3)
        s_c ← s_c ⊕ w_4 round + c
}
```

We need to remember, however, that the $\oplus$ operator here means XORing two column matrices, each of 4 bytes. Writing a simple routine to do that is left as an exercise.

7.3 KEY EXPANSION

To create round key for each round, AES uses a key-expansion process. If the number of rounds is N_r, the **key-expansion** routine creates $N_r + 1$ 128-bit round keys from one single 128-bit cipher key. The first round key is used for pre-round transformation (AddRoundKey); the remaining round keys are used for the last transformation (AddRoundKey) at the end of each round.

The key-expansion routine creates round keys word by word, where a word is an array of four bytes. The routine creates $4 \times (N_r + 1)$ words that are called

$$\mathbf{w}_0, \mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_{4(N_r + 1) - 1}$$

In other words, in the AES-128 version (10 rounds), there are 44 words; in the AES-192 version (12 rounds), there are 52 words; and in the AES-256 version (with 14 rounds), there are 60 words. Each round key is made of four words. Table 7.3 shows the relationship between rounds and words.

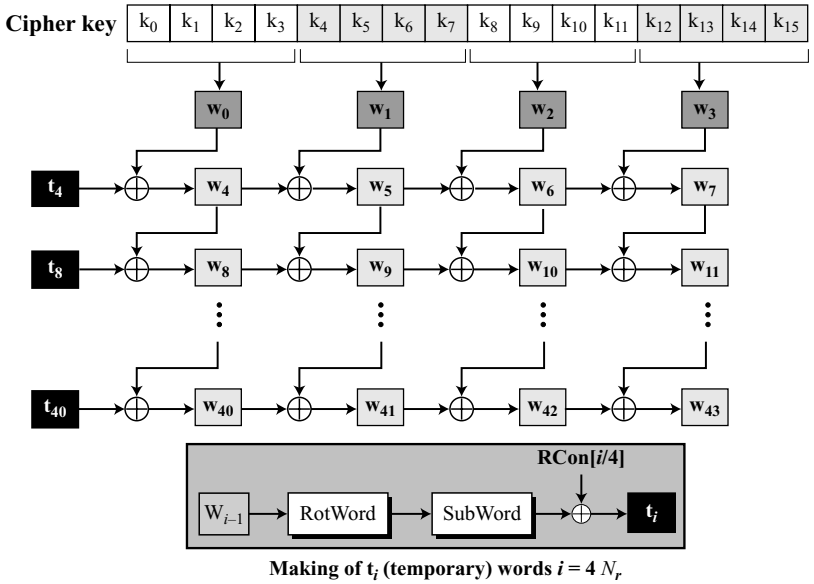
Table 7.3 *Words for each round*

Round	*Words*			
Pre-round	$\mathbf{w}_0$	$\mathbf{w}_1$	$\mathbf{w}_2$	$\mathbf{w}_3$
1	$\mathbf{w}_4$	$\mathbf{w}_5$	$\mathbf{w}_6$	$\mathbf{w}_7$
2	$\mathbf{w}_8$	$\mathbf{w}_9$	$\mathbf{w}_{10}$	$\mathbf{w}_{11}$
. . .	. . .			
N_r	$\mathbf{w}_{4N_r}$	$\mathbf{w}_{4N_r+1}$	$\mathbf{w}_{4N_r+2}$	$\mathbf{w}_{4N_r+3}$

Key Expansion in AES-128

Let us show the creation of words for the AES-128 version; the processes for the other two versions are the same with some slight changes. Figure 7.16 shows how 44 words are made from the original key.

Figure 7.16 *Key expansion in AES*



The process is as follows:

1. The first four words ($\mathbf{w}_0$, $\mathbf{w}_1$, $\mathbf{w}_2$, $\mathbf{w}_3$) are made from the cipher key. The cipher key is thought of as an array of 16 bytes (k_0 to k_{15}). The first four bytes (k_0 to k_3) become $\mathbf{w}_0$; the next four bytes (k_4 to k_7) become $\mathbf{w}_1$; and so on. In other words, the concatenation of the words in this group replicates the cipher key.
2. The rest of the words ($\mathbf{w}_i$ for $i = 4$ to 43) are made as follows:
   a. If (i mod 4) $\neq 0$, $\mathbf{w}_i = \mathbf{w}_{i-1} \oplus \mathbf{w}_{i-4}$. Referring to Figure 7.16, this means each word is made from the one at the left and the one at the top.

b. If (i mod 4) = 0, $\mathbf{w}_i = \mathbf{t} \oplus \mathbf{w}_{i-4}$. Here **t**, a temporary word, is the result of applying two routines, SubWord and RotWord, on $\mathbf{w}_{i-1}$ and XORing the result with a round constants, RCon. In other words, we have,

$$\mathbf{t} = \text{SubWord}\,(\text{RotWord}\,(\mathbf{w}_{i-1})) \oplus \text{RCon}_{i/4}$$

RotWord

The **RotWord** (rotate word) routine is similar to the ShiftRows transformation, but it is applied to only one row. The routine takes a word as an array of four bytes and shifts each byte to the left with wrapping.

SubWord

The **SubWord** (substitute word) routine is similar to the SubBytes transformation, but it is applied only to four bytes. The routine takes each byte in the word and substitutes another byte for it.

Round Constants

Each round constant, RCon, is a 4-byte value in which the rightmost three bytes are always zero. Table 7.4 shows the values for AES-128 version (with 10 rounds).

Table 7.4 *RCon constants*

Round	*Constant (RCon)*	*Round*	*Constant (RCon)*
1	$(\underline{\mathbf{01}}\ 00\ 00\ 00)_{16}$	6	$(\underline{\mathbf{20}}\ 00\ 00\ 00)_{16}$
2	$(\underline{\mathbf{02}}\ 00\ 00\ 00)_{16}$	7	$(\underline{\mathbf{40}}\ 00\ 00\ 00)_{16}$
3	$(\underline{\mathbf{04}}\ 00\ 00\ 00)_{16}$	8	$(\underline{\mathbf{80}}\ 00\ 00\ 00)_{16}$
4	$(\underline{\mathbf{08}}\ 00\ 00\ 00)_{16}$	9	$(\underline{\mathbf{1B}}\ 00\ 00\ 00)_{16}$
5	$(\underline{\mathbf{10}}\ 00\ 00\ 00)_{16}$	10	$(\underline{\mathbf{36}}\ 00\ 00\ 00)_{16}$

The key-expansion routine can either use the above table when calculating the words or use the GF(2^8) field to calculate the leftmost byte dynamically, as shown below (*prime* is the irreducible polynomial):

$$\begin{array}{llllll}
RC_1 & \rightarrow x^{1-1} = x^0 & \text{mod } prime & = 1 & \rightarrow 00000001 & \rightarrow 01_{16} \\
RC_2 & \rightarrow x^{2-1} = x^1 & \text{mod } prime & = x & \rightarrow 00000010 & \rightarrow 02_{16} \\
RC_3 & \rightarrow x^{3-1} = x^2 & \text{mod } prime & = x^2 & \rightarrow 00000100 & \rightarrow 04_{16} \\
RC_4 & \rightarrow x^{4-1} = x^3 & \text{mod } prime & = x^3 & \rightarrow 00001000 & \rightarrow 08_{16} \\
RC_5 & \rightarrow x^{5-1} = x^4 & \text{mod } prime & = x^4 & \rightarrow 00010000 & \rightarrow 10_{16} \\
RC_6 & \rightarrow x^{6-1} = x^5 & \text{mod } prime & = x^5 & \rightarrow 00100000 & \rightarrow 20_{16} \\
RC_7 & \rightarrow x^{7-1} = x^6 & \text{mod } prime & = x^6 & \rightarrow 01000000 & \rightarrow 40_{16} \\
RC_8 & \rightarrow x^{8-1} = x^7 & \text{mod } prime & = x^7 & \rightarrow 10000000 & \rightarrow 80_{16} \\
RC_9 & \rightarrow x^{9-1} = x^8 & \text{mod } prime & = x^4 + x^3 + x + 1 & \rightarrow 00011011 & \rightarrow 1B_{16} \\
RC_{10} & \rightarrow x^{10-1} = x^9 & \text{mod } prime & = x^5 + x^4 + x^2 + x & \rightarrow 00110110 & \rightarrow 36_{16}
\end{array}$$

The leftmost byte, which is called RC_i is actually x^{i-1}, where i is the round number. AES uses the irreducible polynomial $(x^8 + x^4 + x^3 + x + 1)$.

Algorithm

Algorithm 7.5 is a simple algorithm for the key-expansion routine (version AES-128).

Algorithm 7.5 *Pseudocode for key expansion in AES-128*

```
KeyExpansion ([key0 to key15], [w0 to w43])
{
    for (i = 0 to 3)
        wi ← key4i + key4i+1 + key4i+2 + key4i+3

    for (i = 4 to 43)
    {
        if (i mod 4 ≠ 0)    wi ← wi–1 + wi–4
        else
        {
            t ← SubWord (RotWord (wi–1)) ⊕ RCon i/4        // t is a temporary word
            wi ← t + wi–4
        }
    }
}
```

Example 7.6

Table 7.5 shows how the keys for each round are calculated assuming that the 128-bit cipher key agreed upon by Alice and Bob is $(24\ 75\ A2\ B3\ 34\ 75\ 56\ 88\ 31\ E2\ 12\ 00\ 13\ AA\ 54\ 87)_{16}$.

Table 7.5 *Key expansion example*

Round	*Values of t's*	*First word in the round*	*Second word in the round*	*Third word in the round*	*Fourth word in the round*
—		$w_{00} = 2475A2B3$	$w_{01} = 34755688$	$w_{02} = 31E21200$	$w_{03} = 13AA5487$
1	AD20177D	$w_{04} = 8955B5CE$	$w_{05} = BD20E346$	$w_{06} = 8CC2F146$	$w_{07} = 9F68A5C1$
2	470678DB	$w_{08} = CE53CD15$	$w_{09} = 73732E53$	$w_{10} = FFB1DF15$	$w_{11} = 60D97AD4$
3	31DA48D0	$w_{12} = FF8985C5$	$w_{13} = 8CFAAB96$	$w_{14} = 734B7483$	$w_{15} = 2475A2B3$
4	47AB5B7D	$w_{16} = B822deb8$	$w_{17} = 34D8752E$	$w_{18} = 479301AD$	$w_{19} = 54010FFA$
5	6C762D20	$w_{20} = D454F398$	$w_{21} = E08C86B6$	$w_{22} = A71F871B$	$w_{23} = F31E88E1$
6	52C4F80D	$w_{24} = 86900B95$	$w_{25} = 661C8D23$	$w_{26} = C1030A38$	$w_{27} = 321D82D9$
7	E4133523	$w_{28} = 62833EB6$	$w_{29} = 049FB395$	$w_{30} = C59CB9AD$	$w_{31} = F7813B74$
8	8CE29268	$w_{32} = EE61ACDE$	$w_{33} = EAFE1F4B$	$w_{34} = 2F62A6E6$	$w_{35} = D8E39D92$
9	0A5E4F61	$w_{36} = E43FE3BF$	$w_{37} = 0EC1FCF4$	$w_{38} = 21A35A12$	$w_{39} = F940C780$
10	3FC6CD99	$w_{40} = DBF92E26$	$w_{41} = D538D2D2$	$w_{42} = F49B88C0$	$w_{43} = 0DDB4F40$

In each round, the calculation of the last three words are very simple. For the calculation of the first word we need to first calculate the value of temporary word (**t**). For example, the first **t** (for round 1) is calculated as

RotWord (13AA5487) = AA548713 → **SubWord** (AA548713) = AC20177D
t = AC20177D ⊕ $\mathbf{RCon_1}$ = AC20 17 7D ⊕ 01000000_{16} = AD20177D

Example 7.7

Each round key in AES depends on the previous round key. The dependency, however, is nonlinear because of SubWord transformation. The addition of the round constants also guarantees that each round key will be different from the previous one.

Example 7.8

The two sets of round keys can be created from two cipher keys that are different only in one bit.

Cipher Key 1: 12 45 A2 A1 23 31 A4 A3 B2 CC A**A** 34 C2 BB 77 23
Cipher Key 2: 12 45 A2 A1 23 31 A4 A3 B2 CC A**B** 34 C2 BB 77 23

As Table 7.6 shows, there are significant differences between the two corresponding round keys (*R.* means *round* and *B. D.* means *bit difference*).

Table 7.6 *Comparing two sets of round keys*

R.	*Round keys for set 1*	*Round keys for set 2*	*B. D.*
—	1245A2A1 2331A4A3 B2CCAA34 C2BB7723	1245A2A1 2331A4A3 B2CCAB34 C2BB7723	01
1	F9B08484 DA812027 684D8A13 AAF6FD30	F9B08484 DA812027 684D8B13 AAF6FC30	02
2	B9E48028 6365A00F 0B282A1C A1DED72C	B9008028 6381A00F 0BCC2B1C A13AD72C	17
3	A0EAF11A C38F5115 C8A77B09 6979AC25	3D0EF11A 5E8F5115 55437A09 F479AD25	30
4	1E7BCEE3 DDF49FF6 1553E4FF 7C2A48DA	839BCEA5 DD149FB0 8857E5B9 7C2E489C	31
5	EB2999F3 36DD0605 238EE2FA 5FA4AA20	A2C910B5 7FDD8F05 F78A6ABC 8BA42220	34
6	82852E3C B4582839 97D6CAC3 C87260E3	CB5AA788 B487288D 430D4231 C8A96011	56
7	82553FD4 360D17ED A1DBDD2E 69A9BDCD	588A2560 EC0D0DED AF004FDC 67A92FCD	50
8	D12F822D E72295C0 46F948EE 2F50F523	0B9F98E5 E7929508 4892DAD4 2F3BF519	44
9	99C9A438 7EEB31F8 38127916 17428C35	F2794CF0 15EBD9F8 5D79032C 7242F635	51
10	83AD32C8 FD460330 C5547A26 D216F613	E83BDAB0 FDD00348 A0A90064 D2EBF651	52

Example 7.9

The concept of weak keys, as we discussed for DES in Chapter 6, does not apply to AES. Assume that all bits in the cipher key are 0s. The following shows the words for some rounds:

Pre-round:	00000000	00000000	00000000	00000000
Round 01:	62636363	62636363	62636363	62636363
Round 02:	9B9898C9	F9FBFBAA	9B9898C9	F9FBFBAA
Round 03:	90973450	696CCFFA	F2F45733	0B0FAC99
...	...	...	...	...
Round 10:	B4EF5BCB	3E92E211	23E951CF	6F8F188E

The words in the pre-round and the first round are all the same. In the second round, the first word matches with the third; the second word matches with the fourth. However, after the second round the pattern disappears; every word is different.

Key Expansion in AES-192 and AES-256

Key-expansion algorithms in the AES-192 and AES-256 versions are very similar to the key expansion algorithm in AES-128, with the following differences:

1. In AES-192, the words are generated in groups of six instead of four.
   a. The cipher key creates the first six words ($\mathbf{w}_0$ to $\mathbf{w}_5$).
   b. If $i \bmod 6 \neq 0$, $\mathbf{w}_i \leftarrow \mathbf{w}_{i-1} + \mathbf{w}_{i-6}$; otherwise, $\mathbf{w}_i \leftarrow \mathbf{t} + \mathbf{w}_{i-6}$.
2. In AES-256, the words are generated in groups of eight instead of four.
   a. The cipher key creates the first eight words ($\mathbf{w}_0$ to $\mathbf{w}_7$).
   b. If $i \bmod 8 \neq 0$, $\mathbf{w}_i \leftarrow \mathbf{w}_{i-1} + \mathbf{w}_{i-8}$; otherwise, $\mathbf{w}_i \leftarrow \mathbf{t} + \mathbf{w}_{i-8}$.
   c. If $i \bmod 4 = 0$, but $i \bmod 8 \neq 0$, then $\mathbf{w}_i = \text{SubWord}(\mathbf{w}_{i-1}) + \mathbf{w}_{i-8}$.

Key-Expansion Analysis

The key-expansion mechanism in AES has been designed to provide several features that thwart the cryptanalyst.

1. Even if Eve knows only part of the cipher key or the values of the words in some round keys, she still needs to find the rest of the cipher key before she can find all round keys. This is because of the nonlinearity produced by SubWord transformation in the key-expansion process.
2. Two different cipher keys, no matter how similar to each other, produce two expansions that differ in at least a few rounds.
3. Each bit of the cipher key is diffused into several rounds. For example, changing a single bit in the cipher key, will change some bits in several rounds.
4. The use of the constants, the RCons, removes any symmetry that may have been created by the other transformations.
5. There are no serious weak keys in AES, unlike in DES.
6. The key-expansion process can be easily implemented on all platforms.
7. The key-expansion routine can be implemented without storing a single table; all calculations can be done using the GF(2^8) and FG(2) fields.

7.4 CIPHERS

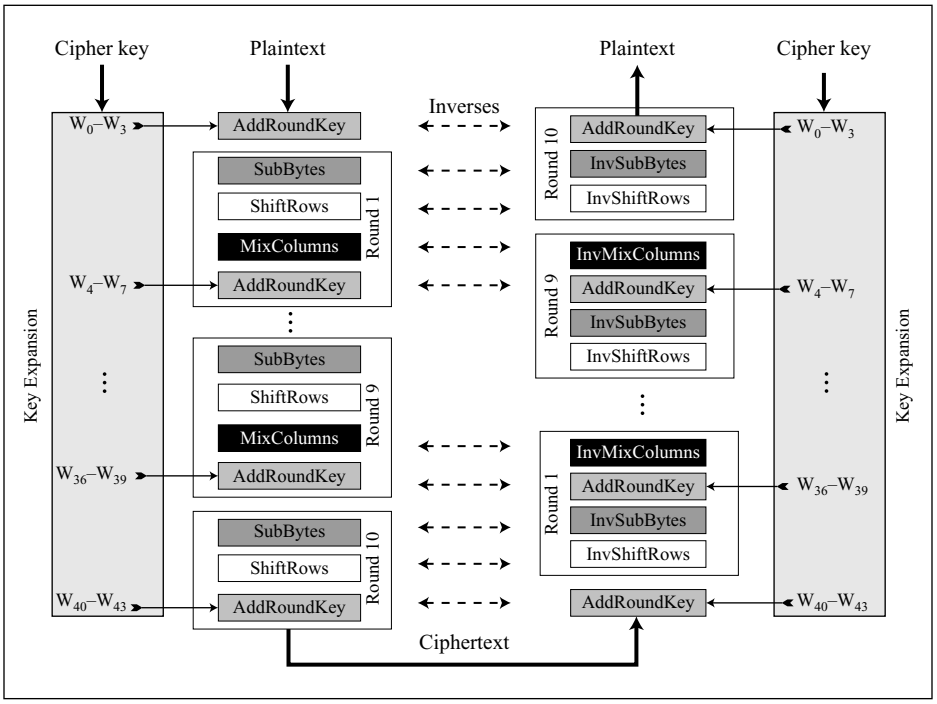
Now let us see how AES uses four types of transformations for encryption and decryption. In the standard, the encryption algorithm is referred to as the **cipher** and the decryption algorithm as the **inverse cipher.**

As we mentioned before, AES is a non-Feistel cipher, which means that each transformation or group of transformations must be invertible. In addition, the cipher and the inverse cipher must use these operations in such a way that cancel each other. The round keys must also be used in the reverse order. Two different designs are given to be used for different implementation. We discuss both designs for AES-128; the designs for other versions are the same.

Original Design

In the original design, the order of transformations in each round is not the same in the cipher and reverse cipher. Figure 7.17 shows this version.

Figure 7.17 *Cipher and inverse cipher of the original design*



First, the order of SubBytes and ShiftRows is changed in the reverse cipher. Second, the order of MixColumns and AddRoundKey is changed in the reverse cipher. This difference in ordering is needed to make each transformation in the

cipher aligned with its inverse in the reverse cipher. Consequently, the decryption algorithm as a whole is the inverse of the encryption algorithm. We have shown only three rounds, but the rest is the same. Note that the round keys are used in the reverse order. Note that the encryption and decryption algorithms in the original design are not similar.

Algorithm

The code for the AES-128 version of this design is shown in Algorithm 7.6. The code for the inverse cipher is left as an exercise.

Algorithm 7.6 *Pseudocode for cipher in the original design*

```
Cipher (InBlock [16], OutBlock[16], w[0 … 43])
{
    BlockToState (InBlock, S)

    S ← AddRoundKey (S, w[0…3])
     for (round = 1 to 10)
    {
        S ← SubBytes (S)
        S ← ShiftRows (S)
         if (round ≠ 10)   S ← MixColumns (S)
         S ← AddRoundKey (S, w[4 × round, 4 × round + 3])
     }

   StateToBlock (S, OutBlock);
}
```

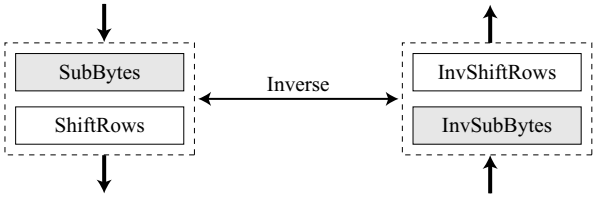
Alternative Design

For those applications that prefer similar algorithms for encryption and decryption, a different inverse cipher was developed. In this version, the transformations in the reverse cipher are rearranged to make the order of transformations the same in the cipher and reverse cipher. In this design, invertibility is provided for a pair of transformations, not for each single transformation.

SubBytes/ShiftRows Pairs

SubBytes change the contents of each byte without changing the order of the bytes in the state; ShiftRows change the order of the bytes in the state without changing the contents of the bytes. This implies that we can change the order of these two transformations in the inverse cipher without affecting the invertibility of the whole algorithm. Figure 7.18 shows the idea. Note that the combination of two transformations in the cipher and inverse cipher are the inverses of each other.

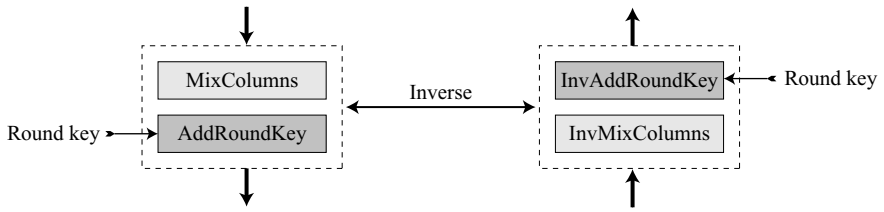
Figure 7.18 *Invertibility of SubBytes and ShiftRows combinations*



MixColumns/AddRoundKey Pair

Here the two involved transformations are of different nature. However, the pairs can become inverses of each other if we multiply the key matrix by the inverse of the constant matrix used in MixColumns transformation. We call the new transformation **InvAddRoundKey.** Figure 7.19 shows the new configuration.

Figure 7.19 *Invertibility of MixColumns and AddRoundKey combinations*



It can be proved that the two combinations are now inverses of each other. In the cipher we call the input state to the combination S and the output state T. In the reverse cipher the input state to the combination is T. The following shows that the output state is also S. Note that the MixColumns transformation is actually multiplication of the C matrix (constant matrix by the state).

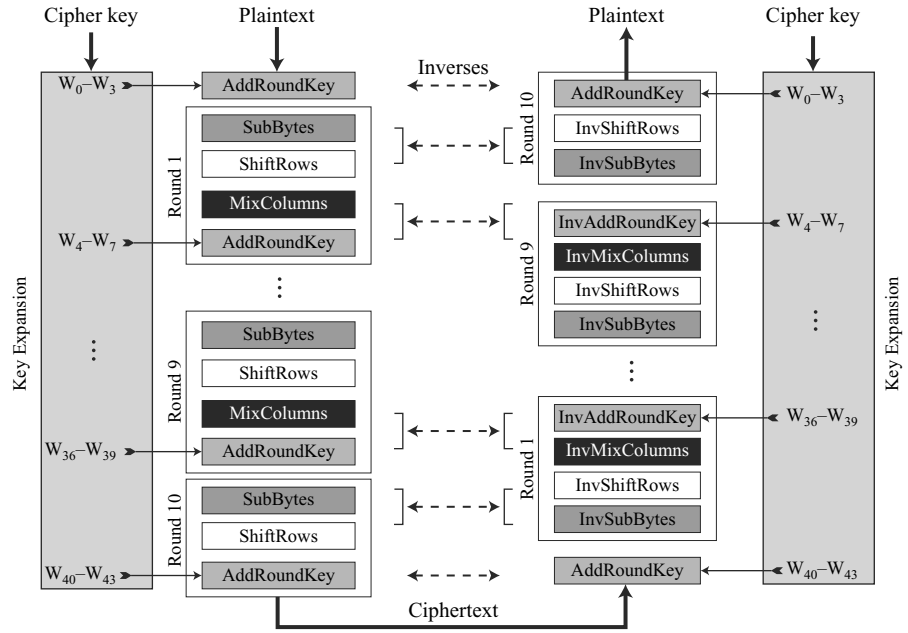
Cipher: $T = CS \oplus K$
Inverse Cipher: $C^{-1}T \oplus C^{-1}K = C^{-1}(CS \oplus K) \oplus C^{-1}K = C^{-1}CS \oplus C^{-1}K \oplus C^{-1}K = S$

Now we can show the cipher and inverse cipher for the alternate design. Note that we still need to use two AddRoundKey transformations in the decryption. In other words, we have nine InvAddRoundKey and two AddRoundKey transformations as shown in Figure 7.20.

Changing Key-Expansion Algorithm

Instead of using InvRoundKey transformation in the reverse cipher, the key-expansion algorithm can be changed to create a different set of round keys for the inverse cipher.

Figure 7.20 *Cipher and reverse cipher in alternate design*



However, note that the round key for the pre-round operation and the last round should not be changed. The round keys for rounds 1 to 9 need to be multiplied by the constant matrix. This algorithm is left as an exercise.

7.5 EXAMPLES

In this section, some examples of encryption/decryption and key generation are given to emphasize some points discussed in the two previous sections.

Example 7.10

The following shows the ciphertext block created from a plaintext block using a randomly selected cipher key.

```
Plaintext:  00 04 12 14 12 04 12 00 0C 00 13 11 08 23 19 19
Cipher Key: 24 75 A2 B3 34 75 56 88 31 E2 12 00 13 AA 54 87
Ciphertext: BC 02 8B D3 E0 E3 B1 95 55 0D 6D FB E6 F1 82 41
```

Table 7.7 shows the values of state matrices and round keys for this example.

Table 7.7 *Example of encryption*

Round	*Input State*	*Output State*	*Round Key*
Pre-round	00 12 0C 08	24 26 3D 1B	24 34 31 13
	04 04 00 23	71 71 E2 89	75 75 E2 AA
	12 12 13 19	B0 44 01 4D	A2 56 12 54
	14 00 11 19	A7 88 11 9E	B3 88 00 87
1	24 26 3D 1B	6C 44 13 BD	89 BD 8C 9F
	71 71 E2 89	B1 9E 46 35	55 20 C2 68
	B0 44 01 4D	C5 B5 F3 02	B5 E3 F1 A5
	A7 88 11 9E	5D 87 FC 8C	CE 46 46 C1
2	6C 44 13 BD	1A 90 15 B2	CE 73 FF 60
	B1 9E 46 35	66 09 1D FC	53 73 B1 D9
	C5 B5 F3 02	20 55 5A B2	CD 2E DF 7A
	5D 87 FC 8C	2B CB 8C 3C	15 53 15 D4
3	1A 90 15 B2	F6 7D A2 B0	FF 8C 73 13
	66 09 1D FC	1B 61 B4 B8	89 FA 4B 92
	20 55 5A B2	67 09 C9 45	85 AB 74 0E
	2B CB 8C 3C	4A 5C 51 09	C5 96 83 57
4	F6 7D A2 B0	CA E5 48 BB	B8 34 47 54
	1B 61 B4 B8	D8 42 AF 71	22 D8 93 01
	67 09 C9 45	D1 BA 98 2D	DE 75 01 0F
	4A 5C 51 09	4E 60 9E DF	B8 2E AD FA
5	CA E5 48 BB	90 35 13 60	D4 E0 A7 F3
	D8 42 AF 71	2C FB 82 3A	54 8C 1F 1E
	D1 BA 98 2D	9E FC 61 ED	F3 86 87 88
	4E 60 9E DF	49 39 CB 47	98 B6 1B E1
6	90 35 13 60	18 0A B9 B5	86 66 C1 32
	2C FB 82 3A	64 68 6A FB	90 1C 03 1D
	9E FC 61 ED	5A EF D7 79	0B 8D 0A 82
	49 39 CB 47	8E B2 10 4D	95 23 38 D9
7	18 0A B9 B5	01 63 F1 96	62 04 C5 F7
	64 68 6A FB	55 24 3A 62	83 9F 9C 81
	5A EF D7 79	F4 8A DE 4D	3E B3 B9 3B
	8E B2 10 4D	CC BA 88 03	B6 95 AD 74
8	01 63 F1 96	2A 34 D8 46	EE EA 2F D8
	55 24 3A 62	2D 6B A2 D6	61 FE 62 E3
	F4 8A DE 4D	51 64 CF 5A	AC 1F A6 9D
	CC BA 88 03	87 A8 F8 28	DE 4B E6 92

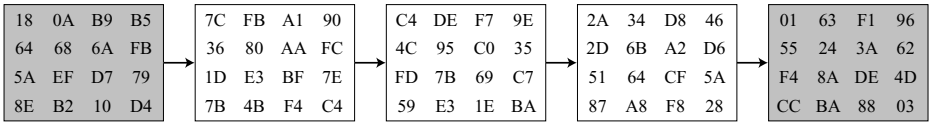
Table 7.7 *Example of encryption (continued)*

Round	*Input State*	*Output State*	*Round Key*
9	2A 34 D8 46	0A D9 F1 3C	E4 0E 21 F9
	2D 6B A2 D6	95 63 9F 35	3F C1 A3 40
	51 64 CF 5A	2A 80 29 00	E3 FC 5A C7
	87 A8 F8 28	16 76 09 77	BF F4 12 80
10	0A D9 F1 3C	BC E0 55 E6	DB D5 F4 0D
	95 63 9F 35	02 E3 0D F1	F9 38 9B DB
	2A 80 29 00	8B B1 6D 82	2E D2 88 4F
	16 76 09 77	D3 95 F8 41	26 D2 C0 40

Example 7.11

Figure 7.21 shows the state entries in one round, round 7, in Example 7.10.

Figure 7.21 *State in a single round*



Example 7.12

One may be curious to see the result of encryption when the plaintext is made of all 0s. Using the cipher key in Example 7.10 yields the ciphertext.

```
Plaintext:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Cipher Key: 24 75 A2 B3 34 75 56 88 31 E2 12 00 13 AA 54 87
Ciphertext: 63 2C D4 5E 5D 56 ED B5 62 04 01 A0 AA 9C 2D 8D
```

Example 7.13

Let us check the avalanche effect that we discussed in Chapter 6. Let us change only one bit in the plaintext and compare the results. We changed only one bit in the last byte. The result clearly shows the effect of diffusion and confusion. Changing a single bit in the plaintext has affected many bits in the ciphertext.

```
Plaintext 1:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Plaintext 2:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01
Ciphertext 1: 63 2C D4 5E 5D 56 ED B5 62 04 01 A0 AA 9C 2D 8D
Ciphertext 2: 26 F3 9B BC A1 9C 0F B7 C7 2E 7E 30 63 92 73 13
```

Example 7.14

The following shows the effect of using a cipher key in which all bits are 0s.

```
Plaintext:   00 04 12 14 12 04 12 00 0c 00 13 11 08 23 19 19
Cipher Key: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Ciphertext: 5A 6F 4B 67 57 B7 A5 D2 C4 30 91 ED 64 9A 42 72
```

7.6 ANALYSIS OF AES

Following is a brief review of the three characteristics of AES.

Security

AES was designed after DES. Most of the known attacks on DES were already tested on AES; none of them has broken the security of AES so far.

Brute-Force Attack

AES is definitely more secure than DES due to the larger-size key (128, 192, and 256 bits). Let us compare DES with 56-bit cipher key and AES with 128-bit cipher key. For DES we need 2^{56}(ignoring the key complement issue) tests to find the key; for AES we need 2^{128} tests to find the key. This means that if we can break DES in t seconds, we need ($2^{72} \times t$) seconds to break AES. This would be almost impossible. In addition, AES provides two other versions with longer cipher keys. The lack of weak keys is another advantage of AES over DES.

Statistical Attacks

The strong diffusion and confusion provided by the combination of the SubBytes, ShiftRows, and MixColumns transformations removes any frequency pattern in the plaintext. Numerous tests have failed to do statistical analysis of the ciphertext.

Differential and Linear Attacks

AES was designed after DES. Differential and linear cryptanalysis attacks were no doubt taken into consideration. There are no differential and linear attacks on AES as yet.

Implementation

AES can be implemented in software, hardware, and firmware. The implementation can use table lookup process or routines that use a well-defined algebraic structure. The transformation can be either byte-oriented or word-oriented. In the byte-oriented version, the whole algorithm can use an 8-bit processor; in the word-oriented version, it can use a 32-bit processor. In either case, the design of constants makes processing very fast.

Simplicity and Cost

The algorithms used in AES are so simple that they can be easily implemented using cheap processors and a minimum amount of memory.

7.7 RECOMMENDED READING

The following books and websites give more details about subjects discussed in this chapter. We recommend the following books and sites. The items enclosed in brackets refer to the reference list at the end of the book.

Books

[Sta06], [Sti06], [Rhe03], [Sal03], [Mao04], and [TW06] discuss AES.

WebSites

The following websites give more information about topics discussed in this chapter.

csrc.nist.gov/publications/fips/fips197/fips-197.pdf

http://www.quadibloc.com/crypto/co040401.htm

http://www.ietf.org/rfc/rfc3394.txt

7.8 KEY TERMS

AddRoundKey
Advanced Encryption Standard (AES)
bit
block
byte
cipher
InvAddRoundKey
inverse cipher
InvMixColumns
InvShiftRows
InvSubBytes
key expansion
MixColumns
National Institute of Standards and Technology (NIST)
Rijndael
RotWord
ShiftRows
state
SubBytes
SubWord
word

7.9 SUMMARY

- The Advanced Encryption Standard (AES) is a symmetric-key block cipher published by NIST as FIPS 197. AES is based on the Rijndael algorithm.

- AES is a non-Feistel cipher that encrypts and decrypts a data block of 128 bits. It uses 10, 12, or 14 number of rounds. The key size, which can be 128, 192, or 256 bits depends on the number of rounds.
- AES is byte-oriented. The 128-bit plaintext or ciphertext is considered as sixteen 8-bit bytes. To be able to perform some mathematical transformations on bytes, AES has defined the concept of a state. A state is a 4×4 matrix in which each entry is a byte.
- To provide security, AES uses four types of transformations: substitution, permutation, mixing, and key-adding. Each round of AES, except the last, uses the four transformations. The last round uses only three of the four transformations.
- Substitution is defined by either a table lookup process or mathematical calculation in the GF(2^8) field. AES uses two invertible transformations, SubBytes and InvSubBytes, which are inverses of each other.
- The second transformation in a round is shifting, which permutes the bytes. In the encryption, the transformation is called ShiftRows. In the decryption, the transformation is called InvShiftRows. The ShiftRows and InvShiftRows transformations are inverses of each other.
- The mixing transformation changes the contents of each byte by taking four bytes at a time and combining them to recreate four new bytes. AES defines two transformations, MixColumns and InvMixColumns, to be used in the encryption and decryption. MixColumns multiplies the state matrix by a constant square matrix; the InvMixColumns does the same using the inverse constant matrix. The MixColumns and InvMixColumns transformations are inverses of each other.
- The transformation that performs whitening is called AddRoundKey. The previous state is added (matrix addition) with the round matrix key to create the new state. Addition of individual elements in the two matrices is done in GF(2^8), which means that 8-bit words are XORed. The AddRoundKey transformation is the inverse of itself.
- In the first configuration (10 rounds with 128-bit keys), the key generator creates eleven 128-bit round keys out of the 128-bit cipher key. AES uses the concept of a word for key generation. A word is made of four bytes. The round keys are generated word by word. AES numbers the words from $\mathbf{w}_0$ to $\mathbf{w}_{43}$. The process is referred to as key expansion.
- AES cipher uses two algorithms for decryption. In the original design, the order of transformations in each round is not the same in the encryption and decryption. In the alternative design, the transformations in the decryption algorithms are rearranged to make ordering the same in encryption and decryption. In the second version, the invertibility is provided for a pair of transformations.

7.10 PRACTICE SET

Review Questions

1. List the criteria defined by NIST for AES.
2. List the parameters (block size, key size, and the number of rounds) for the three AES versions.

3. How many transformations are there in each version of AES? How many round keys are needed for each version?
4. Compare DES and AES. Which one is bit-oriented? Which one is byte-oriented?
5. Define a state in AES. How many states are there in each version of AES?
6. Which of the four transformations defined for AES change the contents of bytes? Which one does not change the contents of the bytes?
7. Compare the substitution in DES and AES. Why do we have only one substitution table (S-box) in AES, but several in DES?
8. Compare the permutations in DES and AES. Why do we need expansion and compression permutations in DES, but not in AES?
9. Compare the round keys in DES and AES. In which cipher is the size of the round key the same as the size of the block?
10. Why do you think the mixing transformation (MixColumns) is not needed in DES, but is needed in AES?

Exercises

11. In a cipher, S-boxes can be either *static* or *dynamic*. The parameters in a static S-box do not depend on the key.
    a. State some advantages and some disadvantages of static and dynamic S-boxes.
    b. Are the S-boxes (substitution tables) in AES static or dynamic?
12. AES has a larger block size than DES (128 versus 64). Is this an advantage or disadvantage? Explain.
13. AES defines different implementations with three different numbers of rounds (10, 12, and 14); DES defines only implementation with 16 rounds. What are the advantages and disadvantages of AES over DES with respect to this difference?
14. AES defines three different cipher-key sizes (128, 192, and 256); DES defines only one cipher-key size (56). What are the advantages and disadvantages of AES over DES with respect to this difference?
15. In AES, the size of the block is the same as the size of the round key (128 bits); in DES, the size of the block is 64 bits, but the size of the round key is only 48 bits. What are the advantages and disadvantages of AES over DES with respect to this difference?
16. Prove that the ShiftRows and InvShiftRows transformations are permutations by doing the following:
    a. Show the permutation table for ShiftRows. The table needs to have 128 entries, but since the contents of a byte do not change, the table can have only 16 entries with the assumption that each entry represents a byte.
    b. Repeat Part a for InvShiftRows transformation.
    c. Using the results of Parts *a* and *b*, prove that the ShiftRows and InvShiftRows transformations are inverses of each other.

17. Using the same cipher key, apply each of the following transformations on two plaintexts that differ only in the first bit. Find the number of bits changed after each transformation. Each transformation is applied independently.
    a. SubBytes
    b. ShiftRows
    c. MixColumns
    d. AddRoundKey (with the same round keys of your choice)
18. To see the nonlinearity of the SubBytes transformation, show that if a and b are two bytes, we have

$$\text{SubBytes}(a \oplus b) \neq \text{SubBytes}(a) \oplus \text{SubBytes}(b)$$

    Use a = 0x57 and b = 0xA2 as an example.
19. Give a general formula to calculate the number of each kind of transformation (SubBytes, ShiftRows, MixColumns, and AddRoundKey) and the number of total transformations for each version of AES. The formula should be parametrized on the number of rounds.
20. Redraw Figure 7.16 for AES-192 and AES-256.
21. Create two new tables that show RCons constants for the AES-192 and AES-256 implementations (see Table 7.4).
22. In AES-128, the round key used in the pre-round operation is the same as the cipher key. Is this the case for AES-192? Is this the case for AES-256?
23. In Figure 7.8, multiply the X and X^{-1} matrices to prove that they are inverses of each other.
24. Using Figure 7.12, rewrite the square matrices C and C^{-1} using polynomials with coefficients in GF(2). Multiply the two matrices and prove that they are inverse of each other.
25. Prove that the code in Algorithm 7.1 (SubBytes transformation) matches the process shown in Figure 7.8.
26. Using Algorithm 7.1 (SubBytes transformation), do the following:
    a. Write the code for a routine that calculates the inverse of a byte in GF(2^8).
    b. Write the code for ByteToMatrix.
    c. Write the code for MatrixToByte.
27. Write an algorithm for the InvSubBytes transformation.
28. Prove that the code in Algorithm 7.2 (ShiftRows transformation) matches the process shown in Figure 7.9.
29. Using Algorithm 7.2 (ShiftRows transformation), write the code for CopyRow routine.
30. Write an algorithm for the InvShiftRows transformation.
31. Prove that the code in Algorithm 7.3 (MixColumns transformation) matches with the process shown in Figure 7.13.

32. Using Algorithm 7.3 (MixColumns transformation), write the code for the CopyColumn routine.
33. Rewrite Algorithm 7.3 (MixColumns transformation) replacing the operators (.) with a routine called MultField to calculate the multiplication of two bytes in the $GF(2^8)$ field.
34. Write an algorithm for InvMixColumns transformation.
35. Prove that the code in Algorithm 7.4 (AddRoundKey transformation) matches the process shown in Figure 7.15.
36. In Algorithm 7.5 (Key Expansion),
    a. Write the code for the SubWord routine.
    b. Write the code for the RotWord routine.
37. Give two new algorithms for key expansion in AES-192 and AES-256 (see Algorithm 7.5).
38. Write the key-expansion algorithm for alternate inverse cipher.
39. Write the algorithm for inverse cipher in the original design.
40. Write the algorithm for the inverse cipher in the alternative design.

APPENDIX P

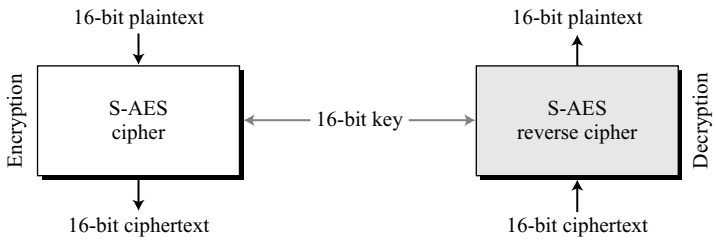
Simplified AES (S-AES)

Simplified AES (S-AES), developed by Professor Edward Schaefer of Santa Clara University, is an educational tool designed to help students learn the structure of AES using smaller blocks and keys. Readers may choose to study this appendix before reading Chapter 7.

P.1 S-AES STRUCTURE

S-AES is a block cipher, as shown in Figure P.1.

Figure P.1 *Encryption and decryption with S-AES*

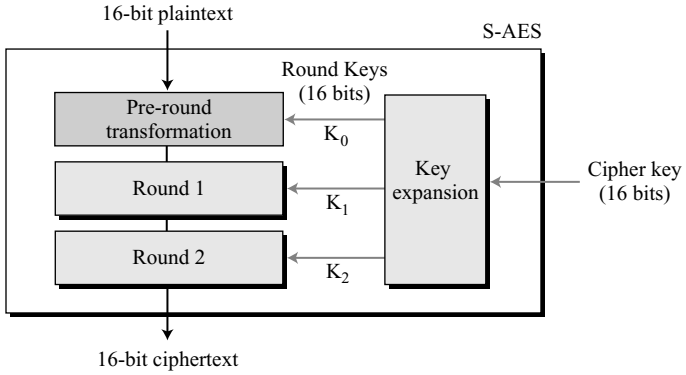


At the encryption site, S-AES takes a 16-bit plaintext and creates a 16-bit ciphertext; at the decryption site, S-AES takes a 16-bit ciphertext and creates a 16-bit plaintext. The same 16-bit cipher key is used for both encryption and decryption.

Rounds

S-AES is a non-Feistel cipher that encrypts and decrypts a data block of 16 bits. It uses one pre-round transformation and two rounds. The cipher key is also 16 bits. Figure P.2

Figure P.2 *General design of S-AES encryption cipher*



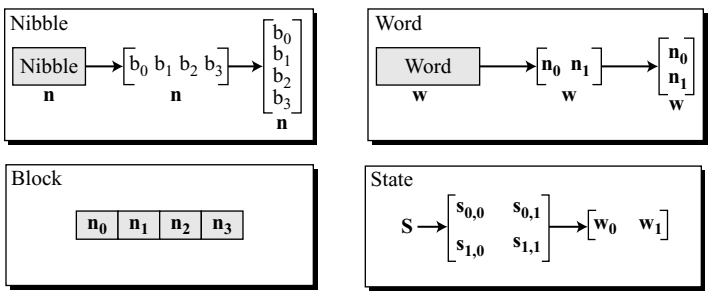
shows the general design for the encryption algorithm (called the cipher); the decryption algorithm (called the inverse cipher) is similar, but the round keys are applied in the reverse order.

In Figure P.2, the round keys, which are created by the key-expansion algorithm, are always 16 bits, the same size as the plaintext or ciphertext block. In S-AES, there are three round keys, K_0, K_1, and K_2.

Data Units

S-AES uses five units of measurement to refer to data: bits, nibbles, words, blocks, and states, as shown in Figure P.3.

Figure P.3 *Data units used in S-AES*



Bit

In S-AES, a *bit* is a binary digit with a value of 0 or 1. We use a lowercase letter b to refer to a bit.

Nibble

A **nibble** is a group of 4 bits that can be treated as a single entity, a row matrix of 4 bits, or a column matrix of 4 bits. When treated as a row matrix, the bits are inserted into the matrix from left to right; when treated as a column matrix, the bits are inserted into the matrix from top to bottom. We use a lowercase bold letter **n** to refer to a nibble. Note that a nibble is actually a single hexadecimal digit.

Word

A *word* is a group of 8 bits that can be treated as a single entity, a row matrix of two nibbles, or a column matrix of 2 nibbles. When it is treated as a row matrix, the nibbles are inserted into the matrix from left to right; when it is considered as a column matrix, the nibbles are inserted into the matrix from top to bottom. We use the lowercase bold letter **w** to refer to a word.

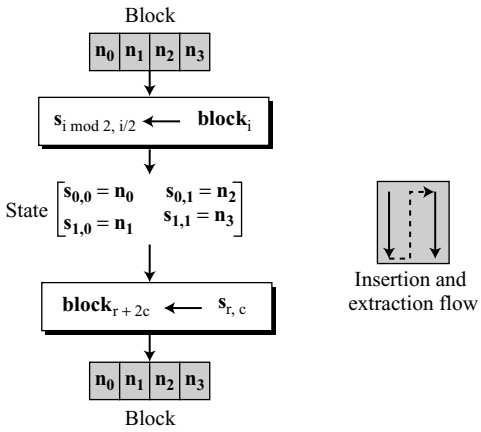
Block

S-AES encrypts and decrypts data blocks. A *block* in S-AES is a group of 16 bits. However, a block can be represented as a row matrix of 4 nibbles.

State

In S-AES, a data block is also referred to as a *state*. We use an uppercase bold letter **S** to refer to a state. States, like blocks, are made of 16 bits, but normally they are treated as matrices of 4 nibbles. In this case, each element of a state is referred to as $\mathbf{s}_{r,c}$, where r (0 to 1) defines the row and the c (0 to 1) defines the column. At the beginning of the cipher, nibbles in a data block are inserted into a state column by column, and in each column, from top to bottom. At the end of the cipher, nibbles in the state are extracted in the same way, as shown in Figure P.4.

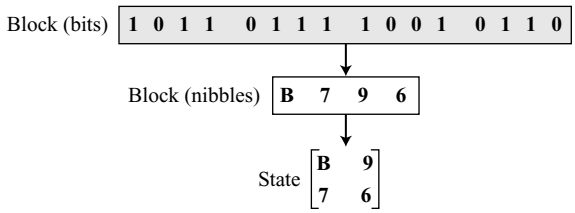
Figure P.4 *Block-to-state and state-to-block transformation*



Example P.1

Let us see how a 16-bit block can be shown as a 2×2 matrix. Assume that the text block is 1011 0111 1001 0110. We first show the block as 4 nibbles. The state matrix is then filled up, column by column, as shown in Figure P.5.

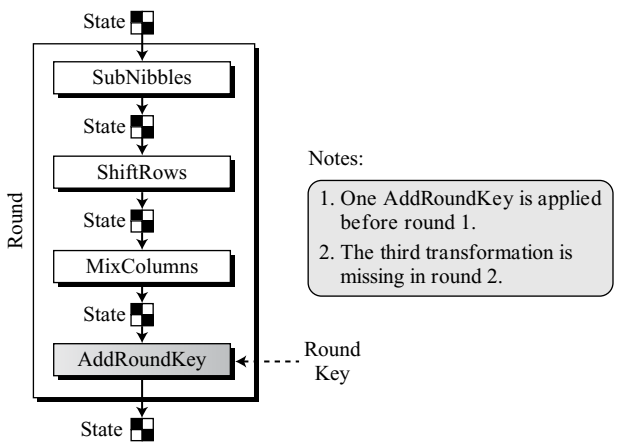
Figure P.5 *Changing ciphertext to a state*



Structure of Each Round

Figure P.6 shows that each transformation takes a state and creates another state to be used for the next transformation or the next round. The pre-round section uses only one transformation (AddRoundKey); the last round uses only three transformations, (MixColumns transformation is missing).

Figure P.6 *Structure of each round at the encryption site*



At the decryption site, the inverse transformations are used: InvSubNibbles, InvShiftRows, InvMixColumns, and AddRoundKey (this one is self-invertible).

P.2 TRANSFORMATIONS

To provide security, S-AES uses four types of transformations: substitution, permutation, mixing, and key-adding. We will discuss each here.

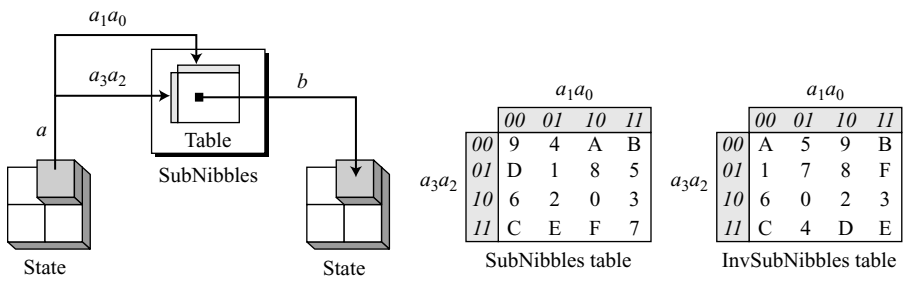
Substitution

Substitution is done for each nibble (4-bit data unit). Only one table is used for transformations of every nibble, which means that if two nibbles are the same, the transformation is also the same. In this appendix, transformation is defined by a table lookup process.

SubNibbles

The first transformation, **SubNibbles,** is used at the encryption site. To substitute a nibble, we interpret the nibble as 4 bits. The left 2 bits define the row and the right 2 bits define the column of the substitution table. The hexadecimal digit at the junction of the row and the column is the new nibble. Figure P.7 shows the idea.

Figure P.7 *SubNibbles transformations*



In the SubNibbles transformation, the state is treated as a 2×2 matrix of nibbles. Transformation is done one nibble at a time. The contents of each nibble is changed, but the arrangement of the nibbles in the matrix remains the same. In the process, each nibble is transformed independently: There are four distinct nibble-to-nibble transformations.

SubNibbles involves four independent nibble-to-nibble transformations.

Figure P.7 also shows the substitution table (S-box) for the SubNibbles transformation. The transformation definitely provides confusion effect. For example, two nibbles, A_{16} and B_{16}, which differ only in one bit (the rightmost bit), are transformed to 0_{16} and 3_{16}, which differ in two bits.

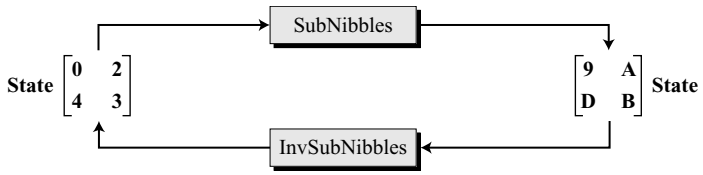
InvSubNibbles

InvSubNibbles is the inverse of SubNibbles. The inverse transformation is also shown in Figure P.7. We can easily check that the two transformations are inverses of each other.

Example P.2

Figure P.8 shows how a state is transformed using the SubNibbles transformation. The figure also shows that the InvSubNibbles transformation creates the original state. Note that if the two nibbles have the same values, their transformation are also the same. The reason is that every nibble uses the same table.

Figure P.8 *SubNibble transformation for Example P.2*



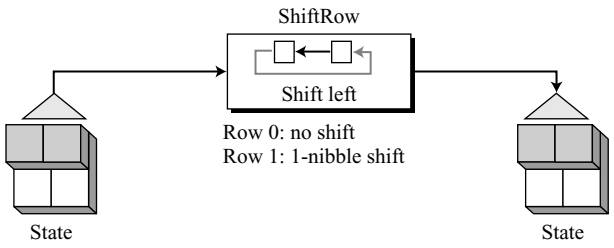
Permutation

Another transformation found in a round is shifting, which permutes the nibbles. Shifting transformation in S-AES is done at the nibble level; the order of the bits in the nibble is not changed.

ShiftRows

In the encryption, the transformation is called *ShiftRows* and the shifting is to the left. The number of shifts depends on the row number (0, 1) of the state matrix. This means row 0 is not shifted at all and row 1 is shifted 1 nibble. Figure P.9 shows the shifting transformation. Note that the ShiftRows transformation operates one row at a time.

Figure P.9 *ShiftRows transformation*



InvShiftRows

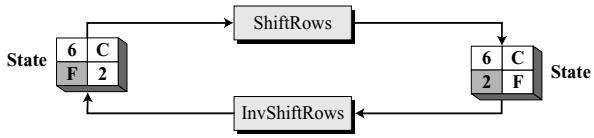
In the decryption, the transformation is called *InvShiftRows* and the shifting is to the right. The number of shifts is the same as the number of the row (0, 1) in the state matrix.

The ShiftRows and InvShiftRows transformations are inverses of each other.

Example P.3

Figure P.10 shows how a state is transformed using ShiftRows. The figure also shows that the InvShiftRows transformation creates the original state.

Figure P.10 *ShiftRows transformation in Example P.3*



Mixing

The substitution provided by the SubNibbles transformation changes the value of the nibble based only on the nibble's original value and an entry in the table; the process does not include the neighboring nibbles. We can say that SubNibbles is an *intra-nibble* transformation. The permutation provided by the ShiftRows transformation exchanges nibbles without permuting the bits inside the bytes. We can say that ShiftRows is a *nibble-exchange* transformation. We also need an *inter-nibble* transformation that changes the bits inside a nibble, based on the bits inside the neighboring nibbles. We need to mix nibbles to provide diffusion at the bit level.

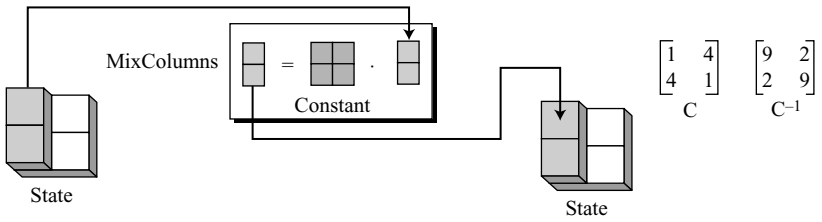
The mixing transformation changes the contents of each nibble by taking 2 nibbles at a time and combining them to create 2 new nibbles. To guarantee that each new nibble is different (even if the old nibbles are the same), the combination process first multiplies each nibble with a different constant and then mixes them. The mixing can be provided by matrix multiplication. As we discussed in Chapter 2, when we multiply a square matrix by a column matrix, the result is a new column matrix. Each element in the new matrix depends on the two elements of the old matrix after they are multiplied by row values in the constant matrix.

MixColumns

The *MixColumns* transformation operates at the column level; it transforms each column of the state into a new column. The transformation is actually the matrix multiplication of a state column by a constant square matrix. The nibbles in the state column and constants matrix are interpreted as 4-bit words (or polynomials) with coefficients in

GF(2). Multiplication of bytes is done in GF(2^4) with modulus ($x^4 + x + 1$) or (10011). Addition is the same as XORing of 4-bit words. Figure P.11 shows the MixColumns transformation.

Figure P.11 *MixColumns transformation*



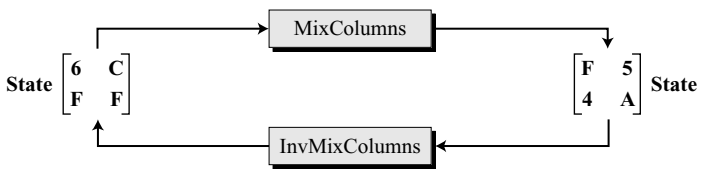
InvMixColumns

The *InvMixColumns* transformation is basically the same as the MixColumns transformation. If the two constant matrices are inverses of each other, it is easy to prove that the two transformations are inverses of each other.

The MixColumns and InvMixColumns transformations are inverses of each other.

Figure P.12 shows how a state is transformed using the MixColumns transformation. The figure also shows that the InvMixColumns transformation creates the original one.

Figure P.12 *The MixColumns transformation in Example 7.5*



Note that equal bytes in the old state, are not equal any more in the new state. For example, the two bytes F in the second row are changed to 4 and A.

Key Adding

Probably the most important transformation is the one that includes the cipher key. All previous transformations use known algorithms that are invertible. If the cipher

key is not added to the state at each round, it is very easy for the adversary to find the plaintext, given the ciphertext. The cipher key is the only secret between Alice and Bob in this case.

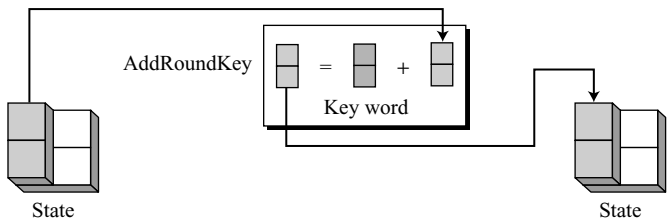
S-AES uses a process called key expansion (discussed later in this appendix) that creates three round keys from the cipher key. Each round key is 16 bits long—it is treated as two 8-bit words. For the purpose of adding the key to the state, each word is considered as a column matrix.

AddRoundKey

AddRoundKey also proceeds one column at a time. It is similar to MixColumns in this respect. MixColumns multiplies a constant square matrix by each state column; AddRoundKey adds a round key word with each state column matrix. The operations in MixColumns are matrix multiplication; the operations in AddRoundKey are matrix addition. The addition is performed in the GF(2^4) field. Because addition and subtraction in this field are the same, the AddRoundKey transformation is the inverse of itself. Figure P.13 shows the AddRoundKey transformation.

The AddRoundKey transformation is the inverse of itself.

Figure P.13 *AddRoundKey transformation*



P.3 KEY EXPANSION

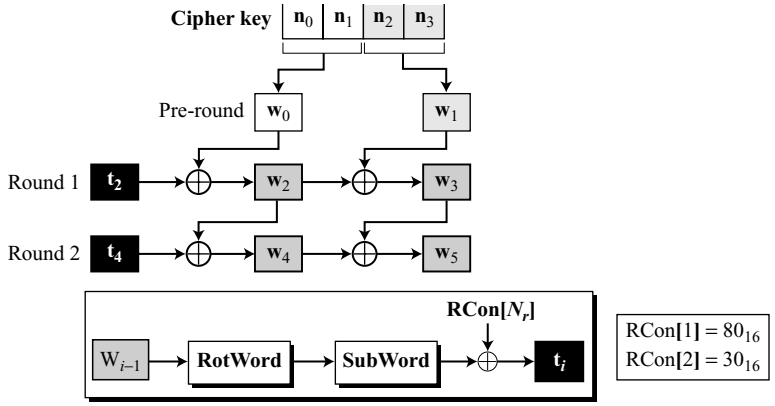
The *key expansion* routine creates three 16-bit round keys from one single 16-bit cipher key. The first round key is used for pre-round transformation (AddRoundKey); the remaining round keys are used for the last transformation (AddRoundKey) at the end of round 1 and round 2.

The key-expansion routine creates round keys word by word, where a word is an array of 2 nibbles. The routine creates 6 words, which are called $\mathbf{w}_0$, $\mathbf{w}_1$, $\mathbf{w}_2$, …, $\mathbf{w}_5$.

Creation of Words in S-AES

Figure P.14 shows how 6 words are made from the original key.

Figure P.14 *Creation of words in S-AES*



Making of $\mathbf{t}_i$ (temporary) words $i = 2N_r$, where N_r is the round number

The process is as follows:

1. The first two words ($\mathbf{w}_0$, $\mathbf{w}_1$) are made from the cipher key. The cipher key is thought of as an array of 4 nibbles ($\mathbf{n}_0$ to $\mathbf{n}_3$). The first 2 nibbles ($\mathbf{n}_0$ to $\mathbf{n}_1$) become $\mathbf{w}_0$; the next 2 nibbles ($\mathbf{n}_2$ to $\mathbf{n}_3$) become $\mathbf{w}_1$. In other words, the concatenation of the words in this group replicates the cipher key.
2. The rest of the words ($\mathbf{w}_i$ for $i = 2$ to 5) are made as follows:
   a. If (i mod 2) = 0, $\mathbf{w}_i = \mathbf{t}_i \oplus \mathbf{w}_{i-2}$. Here $\mathbf{t}_i$, a temporary word, is the result of applying two routines, SubWord and RotWord, on $\mathbf{w}_{i-1}$ and XORing the result with a round constant, RC[N_r], where N_r is the round number. In other words, we have

$$\mathbf{t}_i = \text{SubWord}\,(\text{RotWord}\,(\mathbf{w}_{i-1})) \oplus \text{RCon}\,[N_r]$$

   The words $\mathbf{w}_2$ and $\mathbf{w}_4$ are made using this process.
   b. If (i mod 2) ≠ 0, $\mathbf{w}_i = \mathbf{w}_{i-1} \oplus \mathbf{w}_{i-2}$. Referring to Figure P.14, this means each word is made from the word at the left and the word at the top. The words $\mathbf{w}_3$ and $\mathbf{w}_5$ are made using this process.

RotWord

The *RotWord* (rotate word) routine is similar to the ShiftRows transformation, but it is applied to only one row. The routine takes a word as an array of 2 nibbles and shifts each nibble to the left with wrapping. In S-AES, this is actually swapping the 2 nibbles in the word.

SubWord

The *SubWord* (substitute word) routine is similar to the SubNibble transformation, but it is applied only to 2 nibbles. The routine takes each nibble in the word and substitutes another nibble for it using the SubNibble table in Figure P.7.

Round Constants

Each round constant, RC, is a 2-nibble value in which the rightmost nibble is always zero. Figure P.14 also shows the value of RCs.

Example P.4

Table P.1 shows how the keys for each round are calculated assuming that the 16-bit cipher key agreed upon by Alice and Bob is 2475_{16}.

Table P.1 *Key expansion example*

Round	*Values of t's*	*First word in the round*	*Second word in the round*	*Round Key*
0		$w_0 = 24$	$w_1 = 75$	$K_0 = 2475$
1	$t_2 = 95$	$w_2 = 95 \oplus 24 = B1$	$w_3 = B1 \oplus 75 = C4$	$K_0 = B1C4$
2	$t_4 = EC$	$w_4 = B1 \oplus EC = 5D$	$w_5 = 5D \oplus C4 = 99$	$K_2 = 5D99$

In each round, the calculation of the second word is very simple. For the calculation of the first word we need to first calculate the value of the temporary word ($\mathbf{t}_i$), as shown below:

RotWord (75) = 57 → **SubWord** (57) = 15 → $\mathbf{t_2}$ = 15 ⊕ **RC[1]** = 15 ⊕ 80 = 95

RotWord (C4) = 4C → **SubWord** (4C) = DC → $\mathbf{t_4}$ = DC ⊕ **RC[2]** = DC ⊕ 30 = EC

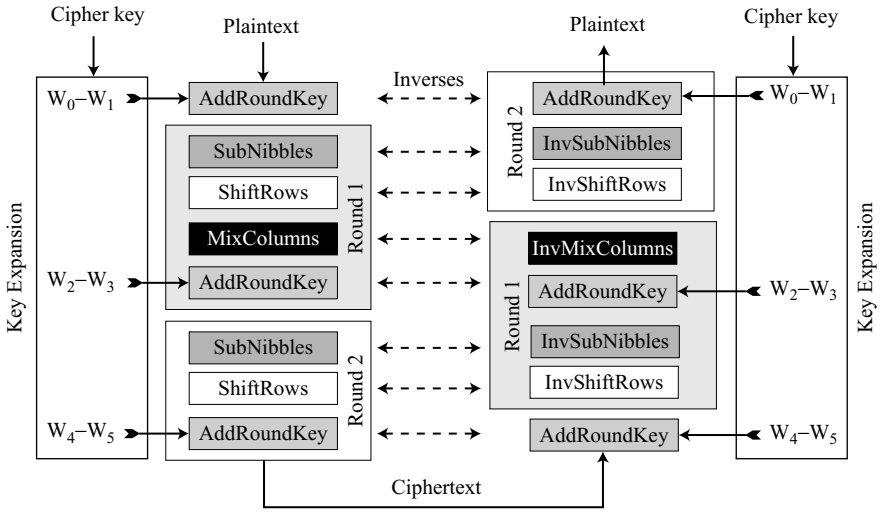
P.4 CIPHERS

Now let us see how S-AES uses the four types of transformations for encryption and decryption. The encryption algorithm is referred to as the *cipher* and the decryption algorithm as the *inverse cipher*.

S-AES is a non-Feistel cipher, which means that each transformation or group of transformations must be invertible. In addition, the cipher and the inverse cipher must use these operations in such a way that they cancel each other. The round keys must also be used in the reverse order. To comply with this requirement, the transformations occur in a different order in the cipher and the reverse cipher, as shown in Figure P.15.

First, the order of SubNibbles and ShiftRows is changed in the reverse cipher. Second, the order of MixColumns and AddRoundKey is changed in the reverse cipher. This difference in ordering is needed to make each transformation in the cipher aligned with its inverse in the reverse cipher. Consequently, the decryption algorithm as a whole is the inverse of the encryption algorithm. Note that the round keys are used in the reverse order.

Figure P.15 *Cipher and inverse cipher of the original design*



Example P.5

We choose a random plaintext block, the cipher key used in Example P.4, and determine what the ciphertext block would be:

Plaintext: $1A23_{16}$	Key: 2475_{16}	Ciphertext: $3AD2_{16}$

Figure P.16 shows the value of states in each round. We are using the round keys generated in Example P.4.

Figure P.16 *Example P.5*