1

Introduction to Block Ciphers

1.1 Block Cipher in Cryptology

1.1.1 Introduction

Information includes our private data that we desire to protect from unwilling leakage depending on the application. **Cryptology** is a field of research that offers appropriate solutions for the data protection by exploring how to construct a secure communication for fair information exchange. Modern cryptology often deals with digitalized data rather than analog data that cannot be expressed simply with a series of 0s and 1s. In our daily life, information is exchanged by digital devices such as radio frequency identification (RFID) tags, smart cards, and smart phones, where a computational resource is limited. Therefore, it is one of the most important challenges in cryptology to realize an efficient implementation of **cryptosystems**.

1.1.2 Symmetric-Key Ciphers

There are various ways to realize **encryption** that is a kind of computational process for information to be protected. In a symmetric-key cipher, information is encrypted with a secret key, and it is expected that the owner of the secret key can decrypt the encrypted information correctly. For instance, let us see the situation, where Alice would like to send a **message** to Bob in a secure way. If the secret key, K, is shared only with Alice and Bob, only Bob can decrypt the message from the encrypted message. The original and the encrypted messages are called **plaintext** and **ciphertext**, respectively. Figure 1.1 illustrates the encryption and decryption processes.

The encryption by Alice can be written as

$$C = E_K(P). \tag{1.1}$$

The ciphertext is decrypted by Bob as

$$P = D_K(C). \tag{1.2}$$

Only Bob can decrypt and read the message, and Eve, who does not own the secret key, cannot decrypt it.

Security of Block Ciphers: From Algorithm Design to Hardware Implementation, First Edition.
Kazuo Sakiyama, Yu Sasaki and Yang Li.
© 2015 John Wiley & Sons Singapore Pte Ltd. Published 2015 by John Wiley & Sons Singapore Pte Ltd.

COPYRIGHTED MATERIAL

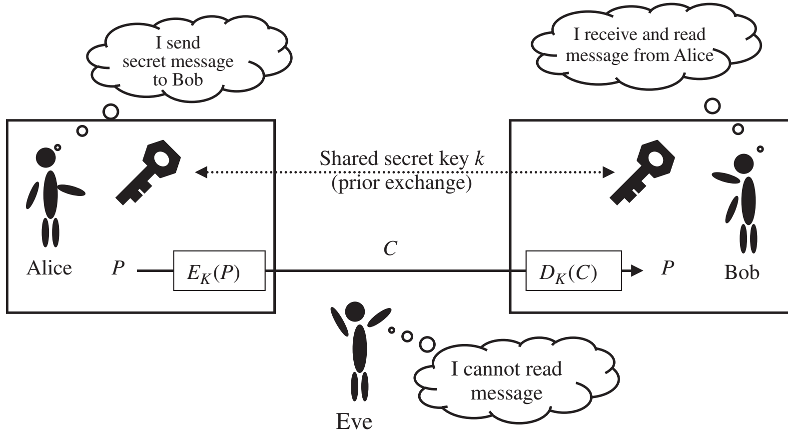

Figure 1.1 Basic model for a symmetric-key cryptosystem

Alice and Bob need to compute the cryptographic operations based on the functions, $E_K(\cdot)$ and $D_K(\cdot)$. The simpler the functions are, the more efficiently they can compute. For instance, **Vernam cipher**, invented in 1917, uses just **XOR** operations as

$$C = P \oplus K, \qquad P = C \oplus K \tag{1.3}$$

to convert plaintext and ciphertext. The XOR operation is explained in Section 1.2.1.

However, in order to guarantee the security, that is, in order that Eve cannot obtain any information of message from C, the secret key needs to be refreshed with a random number for each encryption/decryption. In other words, in order to communicate securely with the Vernam cipher, a very long key, which is the same size as M, is required. This is significantly inefficient. In general, encryption and decryption processes are based on the trade-offs between cost, performance, and security.

1.1.3 Efficient Block Cipher Design

The fundamental idea to achieve an efficient encryption scheme is designing a fixed-input size encryption scheme, and iteratively applying this scheme to encrypt arbitrary length messages. Such a fixed-input size encryption scheme is called **block cipher**, and the group of bits with the fixed-input size is called **block**. If the unit of operation is small enough, for example, 1 bit or 1 byte, such a symmetric-key cipher is called stream cipher. As block ciphers are expected to compute encryption and decryption efficiently, they have an iterated structure, and repeat the same function several times. Such a function is called **round function**. The iterated structure contributes to achieving a small program code in software and implementing a compact circuit design in hardware.

Modern block ciphers are mainly categorized into two kinds: Feistel structure and **substitution-permutation network** (**SPN**) structure. Feistel structure was employed in data encryption standard (DES) block cipher proposed in 1977. Including FEAL and Camellia, the Feistel structure has been employed by many block ciphers.

On the contrary, **Advanced Encryption Standard** (**AES**) employed SPN structure. AES is the main target of this book as it is one of the most widely used block ciphers, and it contains fundamental ideas of SPN structure. The basic mathematics to understand SPN structure and AES specification will be explained later in this chapter.

1.2 Boolean Function and Galois Field

Boolean functions are used in most of the block ciphers including AES. A Boolean function, f, is described as

$$f : \{0,1\}^n \rightarrow \{0,1\}, \tag{1.4}$$

where $\{0,1\}$ is called **Boolean domain** and $\{0,1\}^n$ is the set of all n-tuples $(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are all in Boolean domain.[1]

1.2.1 INV, OR, AND, and XOR Operators

The most simple Boolean function is **inversion** or the **INV** operation that is a bit complement. It operates as

$$\neg x = \begin{cases} 1 & (x = 0), & (1.5a) \\ 0 & (x = 1), & (1.5b) \end{cases}$$

where $\neg$ is used for representing the INV operation. Alternatively, the logic symbol, $^-$, is also used for INV. In this book, we allow both usage, that is, $\neg x = \bar{x}$.

For the case of $n = 2$, representative Boolean functions are **OR**, **AND**, and **XOR**. OR is defined as

$$x \vee y = \begin{cases} 0 & (x = y = 0), & (1.6a) \\ 1 & (\text{else}). & (1.6b) \end{cases}$$

Likewise, AND and XOR are defined, respectively, as

$$x \wedge y = \begin{cases} 1 & (x = y = 1), & (1.7a) \\ 0 & (\text{else}), & (1.7b) \end{cases}$$

$$x \oplus y = \begin{cases} 0 & (x = y), & (1.8a) \\ 1 & (x \neq y). & (1.8b) \end{cases}$$

"$\vee$," "$\wedge$," and "$\oplus$" are used for representing OR, AND, and XOR operations.

The **truth table** for OR, AND, and XOR is described in Table 1.1.

1.2.2 Galois Field

Finite filed or **Galois field** deals with a finite number of elements. Over a Galois filed, addition, subtraction, multiplication, and division are defined. Galois field with the smallest order is

[1] For the case $n = 0$, Boolean function denotes a constant, 0 or 1.

Table 1.1 Truth table for basic operators

x	y	$x \vee y$	$x \wedge y$	$x \oplus y$
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

Table 1.2 Operations over $GF(2)$

x	y	$x + y$	$x \times y$	$-x$	x^{-1}
0	0	0	0	0	–
0	1	1	0	0	–
1	0	1	0	1	1
1	1	0	1	1	1

called a binary field or $GF(2)$. For instance, addition, multiplication, **additive inverse**, and **multiplicative inverse** over $GF(2)$ are defined in Table 1.2.

As can be found from Tables 1.1 and 1.2, addition and multiplication over $GF(2)$ are realized, respectively, with XOR and AND.

Exercise 1.1 *Complete Table 1.3, that is, for addition, multiplication, additive inverse, and multiplicative inverse over* $GF(5)$.

1.2.3 Extended Binary Field and Representation of Elements

Binary field, $GF(2)$, can be extended to a large field size called **extended binary field**, $GF(2^n)$, where n is a positive integer. Especially, in the case of AES, operations in $GF(2^8)$ are of special interest. The number of elements of $GF(2^n)$ is 2^n. There are several different representations for the elements, which affect the cost and speed performance of software and hardware implementations.

1.2.3.1 Polynomial Basis Representation

As the number of elements of $GF(2^n)$ is a power of 2, each bit of the binary representation can be used for each coefficient of a polynomial whose degree is $n - 1$. Any element in $GF(2^n)$ can be expressed with the so-called **polynomial basis** as

$$a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_0, \tag{1.9}$$

Table 1.3 Operations over $GF(5)$

x	y	$x+y$	$x \times y$	$-x$	x^{-1}
0	0				
0	1				
0	2				
0	3				
0	4				
1	0				
1	1				
1	2				
1	3				
1	4				
2	0				
2	1				
2	2				
2	3				
2	4				
3	0				
3	1				
3	2				
3	3				
3	4				
4	0				
4	1				
4	2				
4	3				
4	4				

where $a_i \in \{0,1\}$. For instance, 16 elements in $GF(2^4)$ can be expressed with the binary representation, $(a_3, a_2, a_1, a_0)_2$. By assigning each bit to the coefficient of a polynomial of x, we have $a_3x^3 + a_2x^2 + a_1x + a_0$. Addition of two field elements, for example, $(x+1) + (x^3+1)$, can be calculated as

$$(x+1) + (x^3+1) = x^3 + x, \tag{1.10}$$

as $1+1=0$ over $GF(2)$.

Multiplication of the two field elements, for example, $(x+1)(x^3+1)$, needs modular reduction with an **irreducible polynomial**, for example, x^4+x^3+1, which specifies the field.[2] Therefore, the multiplication result becomes as

$$(x+1)(x^3+1) \equiv x^4 + x^3 + x + 1 \equiv x \ \big(\mathrm{mod}\,(x^4+x^3+1)\big)\,. \tag{1.11}$$

1.2.3.2 Normal Basis Representation

Alternatively, elements in $GF(2^n)$ are described using **normal basis** as

$$b_{n-1}\alpha^{2^{n-1}} + b_{n-2}\alpha^{2^{n-2}} + \cdots + b_0\alpha^{2^0}, \tag{1.12}$$

[2] In this case, we also use the expression, $GF(2)[x]/(x^4+x^3+1)$.

where $b_i \in \{0,1\}$ and α are roots of an irreducible polynomial, $P(x)$, that is,

$$P(\alpha) = 0. \tag{1.13}$$

Furthermore,

$$\alpha^{2^n-1} \equiv 1 \pmod{P(\alpha)}. \tag{1.14}$$

This can be confirmed by Fermat little theorem.

For the case of $GF(2^4)$, suppose that $P(x) = x^4 + x^3 + 1$, that is, $P(\alpha) = \alpha^4 + \alpha^3 + 1 = 0$. Addition in the normal basis representation of $\alpha^7 + \alpha^{11}$ can be calculated simply by XORing each coefficient of two elements in the form of Equation (1.12). That is,

$$\alpha^7 + \alpha^{11} = (\alpha^8 + \alpha^4) + (\alpha^4 + \alpha^2) = \alpha^8 + \alpha^2 = \alpha^{10}, \tag{1.15}$$

where the normal basis representations of α^7 and α^{11} can be found in Table 1.4.

This is correct as $\alpha^7 + \alpha^{11} = \alpha^7(1 + \alpha^4) = \alpha^{10}$. By using the fact of $\alpha^{15} = 1$, multiplication in $GF(2^4)$, for example, $\alpha^7\alpha^{11}$ is calculated as

$$\alpha^7\alpha^{11} = \alpha^{18} = \alpha^3. \tag{1.16}$$

The most advantageous point to use the normal basis representation lies in the fact that **squaring** is easy to compute in $GF(2^n)$. As can be found in Table 1.4, squaring for (b_3, b_2, b_1, b_0) is (b_2, b_1, b_0, b_3). More precisely, in squaring, the elements in the normal basis representation are derived as

$$(b_{n-1}\alpha^{2^{n-1}} + b_{n-2}\alpha^{2^{n-2}} + \cdots + b_0\alpha^{2^0})^2 \tag{1.17}$$

$$= b_{n-1}\alpha^{2^n} + b_{n-2}\alpha^{2^{n-1}} + \cdots + b_0\alpha^{2^1} \tag{1.18}$$

$$= b_{n-2}\alpha^{2^{n-1}} + \cdots + b_0\alpha^{2^1} + b_{n-1}\alpha^{2^0}. \tag{1.19}$$

Table 1.4 Representations of elements for irreducible polynomial $x^4 + x^3 + 1$ in $GF(2^4)$

Binary $(a_3, a_2, a_1, a_0)_2$	Bit concatenation	Hex.	Polynomial basis	Power of α	Normal basis (b_3, b_2, b_1, b_0)
$(0,0,0,0)$	$0\|0\|0\|0$	`0`	0	0	(0, 0, 0, 0)
$(0,0,0,1)$	$0\|0\|0\|1$	`1`	1	1	(1, 1, 1, 1)
$(0,0,1,0)$	$0\|0\|1\|0$	`2`	x	α	(0, 0, 0, 1)
$(0,1,0,0)$	$0\|1\|0\|0$	`4`	x^2	α^2	(0, 0, 1, 0)
$(1,0,0,0)$	$1\|0\|0\|0$	`8`	x^3	α^3	(1, 0, 1, 1)
$(1,0,0,1)$	$1\|0\|0\|1$	`9`	x^3+1	α^4	(0, 1, 0, 0)
$(1,0,1,1)$	$1\|0\|1\|1$	`b`	x^3+x+1	α^5	(0, 1, 0, 1)
$(1,1,1,1)$	$1\|1\|1\|1$	`f`	x^3+x^2+x+1	α^6	(0, 1, 1, 1)
$(0,1,1,1)$	$0\|1\|1\|1$	`7`	x^2+x+1	α^7	(1, 1, 0, 0)
$(1,1,1,0)$	$1\|1\|1\|0$	`e`	x^3+x^2+x	α^8	(1, 0, 0, 0)
$(0,1,0,1)$	$0\|1\|0\|1$	`5`	x^2+1	α^9	(1, 1, 0, 1)
$(1,0,1,0)$	$1\|0\|1\|0$	`a`	x^3+x	α^{10}	(1, 0, 1, 0)
$(1,1,0,1)$	$1\|1\|0\|1$	`d`	x^3+x^2+1	α^{11}	(0, 1, 1, 0)
$(0,0,1,1)$	$0\|0\|1\|1$	`3`	$x+1$	α^{12}	(1, 1, 1, 0)
$(0,1,1,0)$	$0\|1\|1\|0$	`6`	x^2+x	α^{13}	(0, 0, 1, 1)
$(1,1,0,0)$	$1\|1\|0\|0$	`c`	x^3+x^2	α^{14}	(1, 0, 0, 1)

This merit is often used in both software and hardware implementations. However, in general, implementing modular multiplication in the normal basis requires more computation than that in the polynomial basis. Hereafter, we mainly use polynomial basis representation.

1.3 Linear and Nonlinear Functions in Boolean Algebra

1.3.1 Linear Functions

Addition and multiplication by a constant are **linear functions** in $GF(2^n)$. Suppose that $A(x) = a_{n-1}x^{n-1} + \cdots + a_0$ and $B(x) = b_{n-1}x^{n-1} + \cdots + b_0$, where $a_i, b_i \in \{0, 1\}$. Addition of $A(x)$ and $B(x)$ is

$$A(x) + B(x) = (a_{n-1} \oplus b_{n-1})x^{n-1} + \cdots + a_0 \oplus b_0. \quad (1.20)$$

From the fact that $a_i \oplus b_i \in \{0, 1\}$, it is confirmed that addition in $GF(2^n)$ is a linear function.

For multiplication by a constant B, there exist $c_{n-1}, \ldots, c_0 \in \{0, 1\}$ such that

$$A(x) \times B = c_{n-1}x^{n-1} + \cdots + c_0. \quad (1.21)$$

Therefore, we know that such multiplication in $GF(2^n)$ is also a linear function. It can be easily understood considering the fact that multiplication by a constant can be computed with multiple additions of $A(x)$ in $GF(2^n)$.

Exercise 1.2 *Suppose that $A(x) = x^3 + x^2$ and $B(x) = x^3 + x$ are represented in the polynomial basis. Calculate $A(x) + B(x)$, $\mathtt{2}A(x)$, and $\mathtt{3}B(x)$ in $GF(2^4)$ when the irreducible polynomial is $x^4 + x^3 + 1$. Note that* `2` *and* `3` *are hexadecimal representations of x and $x + 1$, respectively.*

Exercise 1.3 *Confirm that modular additive inverse is a linear function.*

1.3.2 Nonlinear Functions

On the contrary, (normal) modular multiplication and multiplicative inverse in $GF(2^n)$ are **nonlinear functions**. The AES block cipher uses a nonlinear function in a part of the design that is based on modular multiplicative inversion in $GF(2)[x]/x^8 + x^4 + x^3 + x + 1$. The multiplicative inverse computation can be done with Fermat's (little) theorem as

$$a^{-1} \equiv a^{2^8-2} \equiv a^{254}, \quad (1.22)$$

for $a \neq 0$. In AES, multiplicative inverse of `0` is mapped to `0`.

One of the most optimal ways to compute the inversion is to find **addition chain**. On the basis of the Itoh–Tsujii algorithm, the computation can be performed with four multiplications and seven modular squarings as

$$\begin{cases} (a)^2 = a^2, & (1.23a) \\ a^2 a = a^3, & (1.23b) \\ (a^3)^{2^2} = a^{12}, & (1.23c) \\ a^{12} a^3 = a^{15}, & (1.23d) \\ (a^{15})^{2^4} = a^{240}, & (1.23e) \\ a^{240} a^2 a^{12} = a^{254}. & (1.23f) \end{cases}$$

Itoh–Tsujii algorithm utilizes the relationship of

$$a^{2^{2^t}-1} = (a^{2^{2^{t-1}}-1})^{2^{2^{t-1}}}(a^{2^{2^{t-1}}-1}). \tag{1.24}$$

1.4 Linear and Nonlinear Functions in Block Cipher

As discussed in Section 1.3, logical operations are classified into linear operations and nonlinear operations. Composition of linear operations is also linear. Hence, if all the cipher's operations are linear, the resulting cipher is also linear, which is insecure. In order to break the linearity of the cipher, nonlinear operations need to be introduced. However, in general, the cost of implementing nonlinear operations is more expensive than the one for linear operations.

The strategy of the block cipher design is alternately applying nonlinear and linear operations several times. To avoid the heavy cost, nonlinear operation is designed to be weak but its cost is small. In many cases, a nonlinear operation is designed to be operated on a smaller size than the block size, and the operation is applied in parallel to all the data. Then, in order to compensate the weak nonlinear computations, a linear operation mixes the entire block. The strategy is depicted in Figure 1.2. In the following, each of the nonlinear layer and linear layer is further detailed.

1.4.1 Nonlinear Layer

In order to reduce the implementation cost, a nonlinear operation is designed to work on a fraction of the data. Typical choices of the size are 64 bits, 32 bits, 8 bits (called byte),

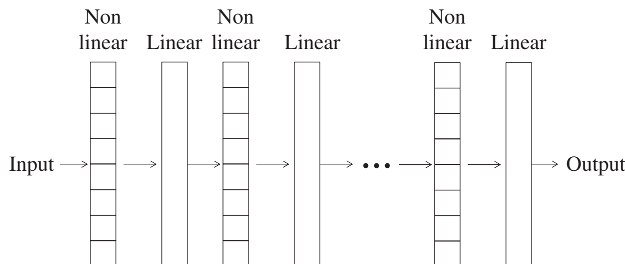

Figure 1.2 Block cipher design strategy. Nonlinear operations and linear operations are alternately applied

Table 1.5 An example of 4-bit to 4-bit S-box, $S(\cdot)$

Input	x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Output	$S(x)$	c	0	f	a	2	b	9	5	8	3	d	7	1	e	6	4

All values are described in the hexadecimal format.

4 bits (called nibble), and 1 bit. The size of the nonlinear operation is determined depending on the following two aspects.

- type of nonlinear operation
- target platform in which the cipher is implemented.

1.4.1.1 Modular Operation

When the cipher is designed for being used in high-end CPUs, the implementation cost is not a big issue but the operation should be optimized for instructions adopted in such a CPU. Currently, many CPUs operate on 64 or 32 bits, thus the size of the nonlinear operation is also adjusted to 64 or 32 bits. The high-end CPUs can perform the modular addition or subtraction efficiently. The nonlinearity is often introduced by addition or subtraction on modulo 2^{64} or 2^{32}.

1.4.1.2 Substitution Table (S-box)

When the cipher is designed for more resource-constrained hardwares such as micro-controllers, the balance of the implementation cost and the computation efficiency is important. When the CPU register size is smaller than 32 bits, the 32- or 64-bit modular addition cannot be performed efficiently. The hardware implementation also faces some problems for those operations. Typical choices of the size of the nonlinear operation are 8 or 4 bits. Because the size is small, using the substitution table is a popular approach to introduce the nonlinearity. The **substitution table**, or **S-box**, is a pre-specified mapping from the input values to the output values. An example of 4-bit to 4-bit S-box is given in Table 1.5.

Exercise 1.4 *Answer the output value of the following computations.*

1. $S(\texttt{2})$
2. $S(\texttt{a})$
3. $S(\texttt{2}) \oplus \texttt{a}$
4. $S(\texttt{2} \oplus \texttt{a})$
5. $S(\texttt{2}) \oplus S(\texttt{a})$
6. $S(S(\texttt{2}) \oplus S(\texttt{a}))$

Exercise 1.5 *Prove that any 1-bit to 1-bit bijective S-box is a linear mapping rather than nonlinear mapping.*

In this S-box, the input value `5` is transformed to `b` according to the table. A 4-bit to 4-bit S-box is implemented only with $16 \times 4 = 64$ bits of memory, which is very small. An 8-bit to 8-bit S-box is implemented only with $256 \times 8 = 2048$ bits of memory, which is bigger than the 4-bit to 4-bit S-box but can mix the data faster than the 4-bit to 4-bit S-box.

1.4.1.3 Boolean Function

A Boolean function is the smallest tool to introduce the nonlinearity. By using an AND or OR operation, the nonlinearity is introduced in 1 bit. When the cipher is designed to be a very resource constraint environment such as RFID, a Boolean function is a typical choice as a source of the nonlinearity. A Boolean function can also fit the high-end CPUs. Thirty two-bit CPUs can operate bit-wise for each of the 32 bits in parallel. If this is combined with modular additions (not bit-wise), the nonlinearity can be introduced quickly.

It is also a popular approach to specify the input and output correspondence of some Boolean functions as an S-box. If the cipher is implemented with some memory, the S-box can be implemented, and the nonlinearity of several bits can be introduced with 1 table look-up. If the cipher is implemented with small hardware, the logic of the Boolean function is implemented to minimize the implementation cost.

1.4.1.4 Balanced Choice

Unfortunately, there is no obvious choice that shows the overwhelming performance in any implementation environment. When the cipher is designed in multi-platforms, that is, both the high- and low-end environment, an S-box maybe chosen as the source of nonlinearity that shows a relatively good performance in both the environments. The popular choices of the nonlinear operations are summarized in Figure 1.3.

Note that the data is mixed by alternately applying a nonlinear operation and a linear operation. The choice of the nonlinear operation also depends on the choice of the linear operation.

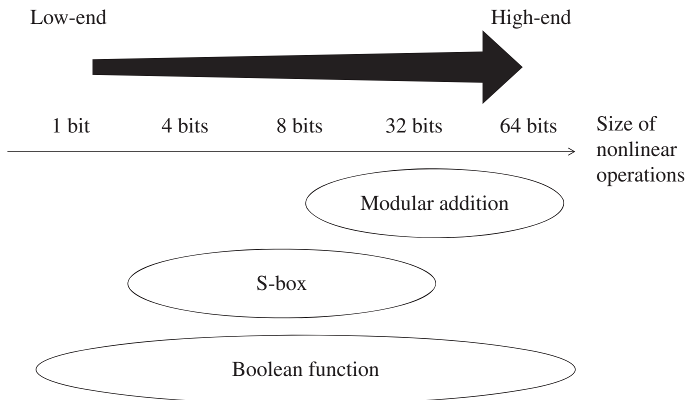

Figure 1.3 Substitution-permutation network. Popular choices of size and type of nonlinear operations

1.4.2 Linear Layer

The purpose of the linear layer is mixing all the output data from the nonlinear layer in which the data is updated in a small part independently. The linear layer is required to be performed efficiently and implemented lightly.

One of the simplest linear operations is XOR. A part of the nonlinear layer output is XORed to another part to mix the data from different parts. The XOR operation can be performed several times between different parts to mix the data more.

The bit-rotation and bit-shift are also simple linear operations. For example, by applying the 1-bit rotation to the entire data, 1-bit from each part will be moved to the next part. The XOR, bit-shift, and bit-rotation can be implemented efficiently in various platforms, thus they are suitable for the block cipher design.

Another important example is a multiplication over a finite field or modular multiplication. Suppose that the size of the nonlinear operation is n bits and each bit of n-bit value represents each coefficient of a polynomial whose degree is $n-1$. As explained in Section 1.3, multiplication over a finite field with some irreducible polynomial $P(x)$ is a linear function. Suppose that the entire data consists of m parts of n-bit data, that is, its size is mn bits. The purpose of the linear function is mixing m independent outputs from the nonlinear layer. In order to mix all the m outputs, $m \times m$ matrices are often used.

For instance, when $m = 4$, four n-bit values x_0, x_1, x_2, x_3 are updated to four n-bit values y_0, y_1, y_2, y_3 by the following matrix operation:

$$\begin{bmatrix} c_0 & c_4 & c_8 & c_{12} \\ c_1 & c_5 & c_9 & c_{13} \\ c_2 & c_6 & c_{10} & c_{14} \\ c_3 & c_7 & c_{11} & c_{15} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}, \tag{1.25}$$

where each c_i is a constant number.

Any combination of linear operations is a linear operation. A popular design approach is combining different types of light linear operations to introduce a strong mixing effect. An example of the linear layer is depicted in Figure 1.4.

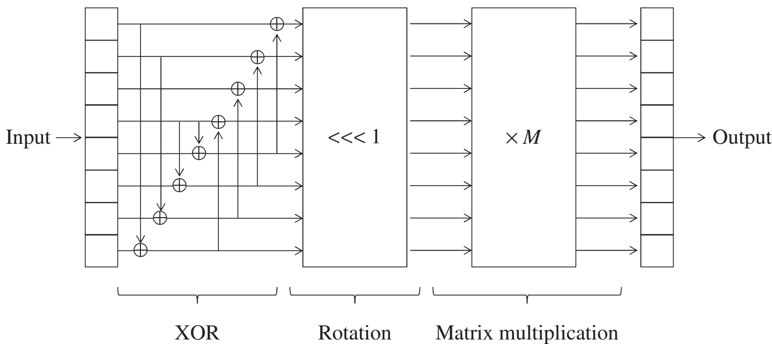

Figure 1.4 An example of linear layer consisting of three linear operations. Nonlinear layer is supposed to update data in eight parts independently

1.4.2.1 Maximum Distance Separable Matrix (MDS Matrix)

A **maximum distance separable** matrix (in short **MDS** matrix) is a matrix with some special property useful for block cipher's design. Considering the usage in block cipher AES, only the case with the same input and output size is discussed here. Let x be the m-component input to the matrix, M, and y be the m-component output from the matrix, that is, $y = Mx$. The matrix M is called MDS if no distinct input-output pairs (x, y) collide in m or more components.

For the application to cryptology, the fact that at least $m + 1$ components differ in distinct pairs of (x, y) is important. In other words, the MDS matrix guarantees a certain amount of change in different input and output values. For instance, suppose that the value of x is slightly modified to x', which differs only 1 bit from x, and the corresponding output value y' is computed. The multiplication by the MDS matrix can guarantee that all the m components of the outputs y and y' have different values, meaning that the 1-bit change of the input always changes all the m components of the output.

1.4.3 Substitution-Permutation Network (SPN)

Substitution-permutation network, which is often called **SPN**, is a design approach to mix a fixed-length input data. SPN is a special form of the iterative application of nonlinear and linear computations.

The substitution layer (or S-layer), which applies a nonlinear operation, is supposed to be an S-box application in a small size. The permutation layer (or P-layer) applies a linear operation to mix the results of the S-layer efficiently.

The SPN structure is adopted in many block ciphers. AES, which is a main target of this book, also adopts the SPN structure.

1.5 Advanced Encryption Standard (AES)

AES is the most widely used block cipher in present time in both governmental and commercial purposes. AES is standardized internationally, and a lot of academic researches and industrial developments have been proposed about AES. This section explains the specification of AES.

The block cipher AES supports three different key sizes: 128 bits, 192 bits, and 256 bits. The corresponding AES algorithms are called **AES-128**, **AES-192**, and **AES-256**, respectively. AES supports a fixed block size: 128 bits. That is to say, when the key is determined, AES provides a bijective map from 128-bit plaintext to 128-bit ciphertext, that is, for a key K, AES-128$_K$, AES-192$_K$, AES-256$_K$:$\{0, 1\}^{128} \rightarrow \{0, 1\}^{128}$ (Figure 1.5).

1.5.1 Specification of AES-128 Encryption

In high level, the 128-bit key K is expanded to eleven 128-bit **subkeys** $sk_0, sk_1, \ldots, sk_{10}$ according to the **key schedule function**, or **KSF**.

1. The 128-bit key K is set to the first 128-bit subkey sk_0.
2. The KSF is computed to update 128-bit subkey sk_0 to another 128-bit subkey sk_1.

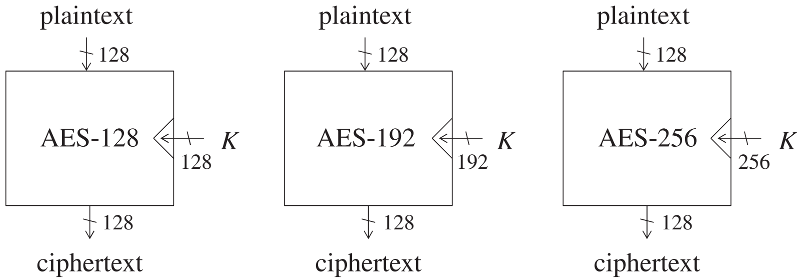

Figure 1.5 Three algorithms of AES

3. Similarly, the KSF is iterated nine times. In each time, 128-bit subkey sk_{i-1} is updated to another 128-bit subkey sk_i for $i = 2, 3, \ldots, 10$.

Then, a plaintext is encrypted to a ciphertext as follows:

1. An XOR of the plaintext and the first subkey sk_0 is computed, and this value is set to a 128-bit internal state value $state_1$. This operation is often called **whitening**.
2. The 128-bit internal state value $state_1$ is updated to $state_2$ by computing a round function, which also takes as input subkey sk_1. This operation is called **round 1** or **the first round**.
3. The round function is iterated nine times to update the internal state value $state_2$ to $state_3, state_4, \ldots, state_{11}$. In round i, where $i = 2, 3, \ldots, 10$, the round function takes as input $(state_i, sk_i)$ and outputs $state_{i+1}$. Note that the round function in the last round is slightly different from the other rounds. The last state that is $state_{11}$ is the ciphertext.

The computation structure of AES-128 in a function level is described in Figure 1.6.

In practice, it is not necessary to compute all the 11 subkeys at the very beginning. For example, the last subkey will not be used until the very end of the encryption process. Thus, generating the last subkey and keeping it in a register is a waste of computation resource. In order to minimize the computation resource, the KSF and the round function updates are computed in parallel round by round. The AES-128 encryption algorithm in the function level can be described as Algorithm 1.1.

1.5.1.1 Preliminaries to Describe Computation Details

In AES, **byte** represents 8-bit values. AES is a byte-oriented cipher. All operations are defined at byte level. Let v be a byte value and $v_7 \| v_6 \| v_5 \| v_4 \| v_3 \| v_2 \| v_1 \| v_0$ be its bit-wise representation, of which the corresponding vector representation is $(v_7 v_6 v_5 v_4 v_3 v_2 v_1 v_0)_2$. In AES, each bit of a byte represents coefficients of polynomial of $GF(2^8)$:

$$v_7 x^7 + v_6 x^6 + v_5 x^5 + v_4 x^4 + v_3 x^3 + v_2 x^2 + v_1 x + v_0. \tag{1.26}$$

A byte value can be represented in hexadecimal. For example, the byte `9b` represents the polynomial $x^7 + x^4 + x^3 + x + 1$.

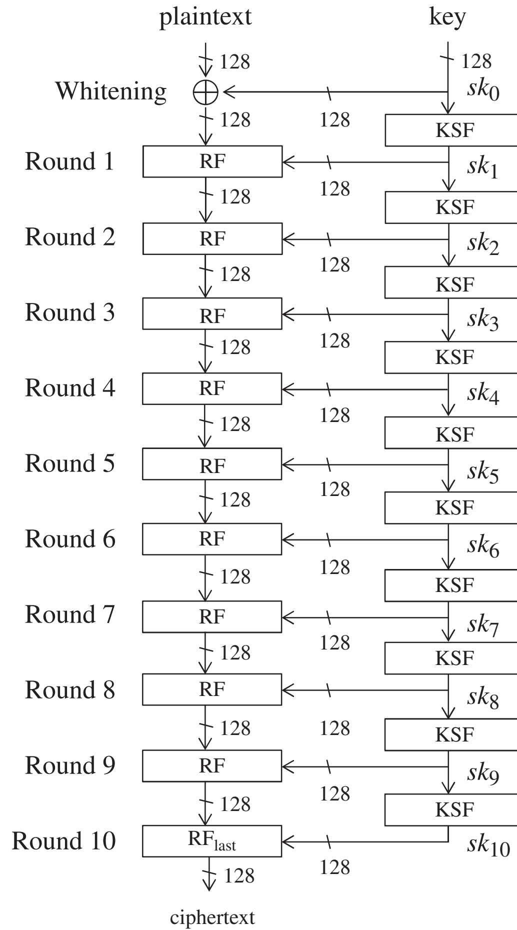

Figure 1.6 High-level computation structure of the encryption of AES-128. RF and KSF denote the round function and KSF, respectively. $\mathrm{RF}_{\mathrm{last}}$ is the last round function, which is different from the other rounds

---

Algorithm 1.1 AES-128 Encryption Algorithm in the Function Level

---

Input: Plaintext P, 128-bit key K, round function RF, the last round function $\mathrm{RF}_{\mathrm{last}}$, key schedule function KSF
Output: Ciphertext C
1: $sk_0 \leftarrow K$;
2: $state_1 \leftarrow P \oplus sk_0$;
3: **for** $i = 1, 2, \ldots, 9$ **do**
4: $\quad sk_i \leftarrow \mathrm{KSF}(sk_{i-1})$;
5: $\quad state_{i+1} \leftarrow \mathrm{RF}(state_i, sk_i)$;
6: **end for**
7: $C \leftarrow \mathrm{RF}_{\mathrm{last}}(state_{10}, sk_{10})$;
8: **return** C;

Addition

Addition of two bytes, $v_7\|v_6\|v_5\|v_4\|v_3\|v_2\|v_1\|v_0$ and $u_7\|u_6\|u_5\|u_4\|u_3\|u_2\|u_1\|u_0$, returns

$$(v_7 \oplus u_7)\|(v_6 \oplus u_6)\|(v_5 \oplus u_5)\|(v_4 \oplus u_4)\|(v_3 \oplus u_3)\|(v_2 \oplus u_2)\|(v_1 \oplus u_1)\|(v_0 \oplus u_0). \tag{1.27}$$

Multiplication

Multiplication in $GF(2^8)$ corresponds with multiplication of polynomials modulo, an irreducible binary polynomial of degree 8. The irreducible polynomial of AES is defined as

$$P(x) = x^8 + x^4 + x^3 + x + 1. \tag{1.28}$$

Because the multiplication by $v(x) \cdot \texttt{02}$ and $v(x) \cdot \texttt{03}$ is later introduced inside the round function, more details of the operation $v(x) \cdot \texttt{02}$ are explained here. $v(x) \cdot \texttt{02}$ is written as

$$(v_7x^7 + v_6x^6 + v_5x^5 + v_4x^4 + v_3x^3 + v_2x^2 + v_1x + v_0) \cdot x \tag{1.29}$$

$$= v_7x^8 + v_6x^7 + v_5x^6 + v_4x^5 + v_3x^4 + v_2x^3 + v_1x^2 + v_0x. \tag{1.30}$$

When $v_7 = 0$, the result is $v_6\|v_5\|v_4\|v_3\|v_2\|v_1\|v_0\|0$ according to the definition of byte. When $v_7 = 1$, the irreducible polynomial $P(x)$ is subtracted from the result. Subtraction is the inverse of the addition. Because the addition is the XOR, the subtraction is also a simple application of the XOR operations. Hence, the result is

$$(v_6x^7 + v_5x^6 + v_4x^5 + v_3x^4 + v_2x^3 + v_1x^2 + v_0x) \oplus (x^4 + x^3 + x + 1) \tag{1.31}$$

$$= v_6x^7 + v_5x^6 + v_4x^5 + (v_3 \oplus 1)x^4 + (v_2 \oplus 1)x^3 + v_1x^2 + (v_0 \oplus 1)x + 1. \tag{1.32}$$

According to the definition of byte, the result is $v_6\|v_5\|v_4\|\bar{v_3}\|\bar{v_2}\|v_1\|\bar{v_0}\|1$.

1.5.1.2 S-box

AES uses a substitution-box (**S-box**) to mix the data. The S-box is used in both of the round function and the KSF, and thus is defined here. The S-box used in AES is a pre-determined bijective mapping from an 8-bit value to an 8-bit value. The definition of the AES S-box is shown in Table 1.6. Hereafter, the S-box transformation is described as $S(\cdot)$. For example, $S(\texttt{4e})$ returns `2f`, and $S(\texttt{d5})$ returns `03`.

Note that the S-box and the inverse S-box transformations are not identical. As explained later, AES decryption algorithm requires the look-up table for the inverse of $S(\cdot)$, that is $S^{-1}(\cdot)$.

1.5.1.3 State

The block size of AES is 128 bits. In AES, 128-bit data is called **state**. The 128-bit state consists of 16 bytes, and is represented as a 4×4 two-dimensional array as depicted in Figure 1.7.

Table 1.6 AES S-box

		Lower four digits															
		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
Upper	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
four digits	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

* All the numbers in this table are in hexadecimal.

State S

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

Figure 1.7 AES state. Each cell denotes a byte

1.5.1.4 Key Schedule Function (KSF)

The 128-bit key K is loaded into a 128-bit subkey sk_0. Then, $sk_i \leftarrow \mathrm{KSF}(sk_{i-1})$ is computed for $i = 1, 2, \ldots, 10$. The input sk_{i-1} is represented as a state. The state is further divided into four columns: $sk_{i-1}(Col(0))$, $sk_{i-1}(Col(1))$, $sk_{i-1}(Col(2))$, and $sk_{i-1}(Col(3))$. The output sk_i is computed column by column. At first, a temporary 4-byte value `tmp` is generated by using the value of $sk_{i-1}(Col(3))$.

1. `tmp` $\leftarrow sk_{i-1}(Col(3))$.
2. Apply the S-box defined in Table 1.6 to each of the 4 bytes in `tmp`.
3. Rotate `tmp` by 1 byte. Precisely, let $\mathtt{tmp}_0 \| \mathtt{tmp}_1 \| \mathtt{tmp}_2 \| \mathtt{tmp}_3$ be the 4 bytes of `tmp`. Then, `tmp` is updated to $\mathtt{tmp}_1 \| \mathtt{tmp}_2 \| \mathtt{tmp}_3 \| \mathtt{tmp}_0$.
4. XOR the pre-specified 1-byte constant $\mathrm{rcon}(i)$ to the first byte of `tmp`.

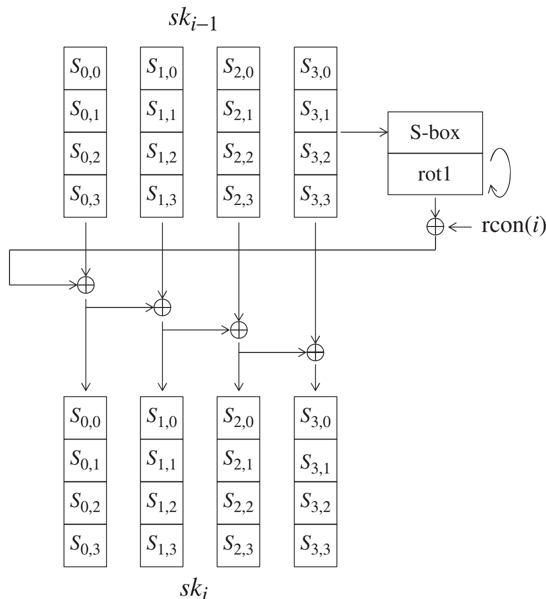

Figure 1.8 Key schedule function of AES-128. The key schedule function is iterated for $i = 1, 2, \ldots, 10$

Then, by using the 4-byte value `tmp`, the next subkey sk_i is generated as follows.

1. $sk_i(Col(0)) \leftarrow \texttt{tmp} \oplus sk_{i-1}(Col(0))$.
2. $sk_i(Col(1)) \leftarrow sk_i(Col(0)) \oplus sk_{i-1}(Col(1))$.
3. $sk_i(Col(2)) \leftarrow sk_i(Col(1)) \oplus sk_{i-1}(Col(2))$.
4. $sk_i(Col(3)) \leftarrow sk_i(Col(2)) \oplus sk_{i-1}(Col(3))$.

The key schedule procedure for AES-128 is depicted in Figure 1.8.

Exercise 1.6 *Write the algorithm of the key schedule function for AES-128. The similar style as Algorithm 1.1 can be used.*

1.5.1.5 Round Function (RF)

The round function takes as input the previous 128-bit state $state_i$ and subkey sk_i, and generates the next 128-bit state $state_{i+1}$. The round function consists of four transformations called **SubBytes**, **ShiftRows**, **MixColumns**, and **AddRoundKey**. It updates the state by following Algorithm 1.2.

Algorithm 1.2 AES Round Function

Input: Previous State S_i, subkey sk_i
Output: New State S_{i+1}
1: Set temporary state $S_{tmp} \leftarrow S_i$;
2: $S_{tmp} \leftarrow \mathrm{SubBytes}(S_{tmp})$;
3: $S_{tmp} \leftarrow \mathrm{ShiftRows}(S_{tmp})$;
4: $S_{tmp} \leftarrow \mathrm{MixColumns}(S_{tmp})$;
5: $S_{tmp} \leftarrow S_{tmp} \oplus sk_i$;
6: $S_{i+1} \leftarrow S_{tmp}$;
7: **return** S_{i+1};

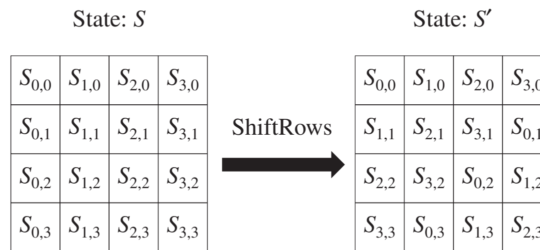

Figure 1.9 ShiftRows operation

SubBytes (SB)
SubBytes is a byte-wise operation. It updates the state by applying the S-box defined in Table 1.6 to each of the 16 bytes of the state. It is worth noting that the SubBytes operation is the only nonlinear one in the AES round function.

ShiftRows (SR)
ShiftRows is a row-wise byte positions swap. The state consists of four rows: row 0, row 1, row 2, and row 3. Each row of the state consists of 4 bytes. The ShiftRows operation applies a left cyclic shift by i bytes to the 4 bytes of row i. The ShiftRows operation is depicted in Figure 1.9.

MixColumns (MC)
MixColumns is a column-wise data mixing operation. It takes as input 4 bytes in a column and computes another 4-byte value. The same computation is applied to all of the four columns. Let x_0, x_1, x_2, x_3 and y_0, y_1, y_2, y_3 be the 4-byte input and 4-byte output, respectively. The y_0, y_1, y_2, y_3 is computed by the following matrix operation:

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}. \tag{1.33}$$

Each element in the matrix is written in hexadecimal.

The MixColumns operation was designed to satisfy the MDS property explained in Section 1.4.2.1. The impact of modifying 1 input byte always expands to all the 4 output bytes. More generally, the sum of the number of modified input bytes and the number of modified output bytes is always greater than or equal to 5.

AddRoundKey (AK)

AddRoundKey updates the state by XORing the subkey sk_i to the state.

Last Round Function ($\mathbf{RF_{last}}$)

In the last round (Round 10 for AES-128), the round function is different from the middle rounds. The MixColumns operation is not computed that is, only the SubBytes, ShiftRows, and AddRoundKey operations are performed.

Exercise 1.7 *Let us consider exchanging the order of two operations in the round function. Which of the following choices return the same result as the original AES specification even if the operations order is exchanged? Why do they return the same result?*

1. SubBytes *and* ShiftRows
2. ShiftRows *and* MixColumns
3. MixColumns *and* AddRoundKey

1.5.2 AES-128 Decryption

To decrypt ciphertext C to P, the round function is applied in reverse order. The KSF is exactly the same. Eleven subkeys $sk_0, sk_1, \dots, sk_{10}$ are generated from K. Different from the encryption algorithm, sk_{10} is firstly used, and then the decryption is processed with $sk_9, sk_8, \dots$, and finally with sk_0.

Inside the round function, four operations are computed in reverse order. The inverse of the AddRoundKey operation is exactly the same as the original AddRoundKey operation because the XOR operation is involution.

For the inverse of the MixColumns operation, the inversion matrix is required. Let b_0, b_1, b_2, b_3 and a_0, a_1, a_2, a_3 be the 4-byte input and 4-byte output to the inverse MixColumns operation, respectively. The a_0, a_1, a_2, a_3 is computed by the following matrix operation:

$$\begin{bmatrix} \mathtt{e} & \mathtt{b} & \mathtt{d} & \mathtt{9} \\ \mathtt{9} & \mathtt{e} & \mathtt{b} & \mathtt{d} \\ \mathtt{d} & \mathtt{9} & \mathtt{e} & \mathtt{b} \\ \mathtt{b} & \mathtt{d} & \mathtt{9} & \mathtt{e} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}. \tag{1.34}$$

Each element in the matrix is again written in hexadecimal.

Table 1.7 AES inverse S-box

		Lower four digits															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Upper four digits	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

All the numbers in this table are in hexadecimal.

The inverse of the ShiftRows operation is relatively simple. It applies a right cyclic shift by i bytes to the 4 bytes of row i.

The inverse of the SubBytes operation requires another table to substitute each byte value. The inverse S-box, denoted by $S(\cdot)^{-1}$, is defined in Table 1.7.

Exercise 1.8 *Write the AES-128 decryption algorithm. The similar style as Algorithm 1.1 can be used.*

1.5.3 Specification of AES-192 and AES-256

AES supports not only the 128-bit key but also the 192-bit and the 256-bit keys. For all the key sizes, round function is identical. The differences are the number of rounds computed and the KSF.

- AES-128 generates eleven 128-bit subkeys $sk_0, sk_1, \ldots, sk_{10}$ from 128-bit K, and the number of rounds is 10.
- AES-192 generates thirteen 128-bit subkeys $sk_0, sk_1, \ldots, sk_{12}$ from 192-bit K, and the number of rounds is 12.
- AES-256 generates fifteen 128-bit subkeys $sk_0, sk_1, \ldots, sk_{14}$ from 256-bit K, and the number of rounds is 14.

1.5.3.1 The Key Schedule Function for AES-192

The 192-bit key K is loaded into a 4×6 array of bytes, which is denoted by $Kstate_0$. Then, $Kstate_i \leftarrow \mathrm{KSF}(Kstate_{i-1})$ is computed for $i = 1, 2, \ldots, 8$. The state is further divided into six columns: $Kstate_{i-1}(Col(0))$, $Kstate_{i-1}(Col(1))$, $Kstate_{i-1}(Col(2))$, $Kstate_{i-1}(Col(3))$, $Kstate_{i-1}(Col(4))$, and $Kstate_{i-1}(Col(5))$. The output $Kstate_i$ is computed column by column. At first, a temporary 4-byte value `tmp` is generated by using the value of $Kstate_{i-1}(Col(5))$.

1. $\texttt{tmp} \leftarrow Kstate_{i-1}(Col(5))$.
2. Apply the S-box defined in Table 1.6 to each of the 4 bytes in `tmp`.
3. Rotate `tmp` by 1 byte. Precisely, let $\texttt{tmp}_0 \| \texttt{tmp}_1 \| \texttt{tmp}_2 \| \texttt{tmp}_3$ be the 4 bytes of `tmp`. Then, `tmp` is updated to $\texttt{tmp}_1 \| \texttt{tmp}_2 \| \texttt{tmp}_3 \| \texttt{tmp}_0$.
4. XOR the pre-specified 1-byte constant $\mathrm{rcon}(i)$ to the first byte of `tmp`.

Then, by using the 4-byte value `tmp`, the next subkey $Kstate_i$ is generated as follows.

1. $Kstate_i(Col(0)) \leftarrow \texttt{tmp} \oplus Kstate_{i-1}(Col(0))$.
2. $Kstate_i(Col(1)) \leftarrow Kstate_i(Col(0)) \oplus Kstate_{i-1}(Col(1))$.
3. $Kstate_i(Col(2)) \leftarrow Kstate_i(Col(1)) \oplus Kstate_{i-1}(Col(2))$.
4. $Kstate_i(Col(3)) \leftarrow Kstate_i(Col(2)) \oplus Kstate_{i-1}(Col(3))$.
5. $Kstate_i(Col(4)) \leftarrow Kstate_i(Col(3)) \oplus Kstate_{i-1}(Col(4))$.
6. $Kstate_i(Col(5)) \leftarrow Kstate_i(Col(4)) \oplus Kstate_{i-1}(Col(5))$.

Among the 192-bit of the $Kstate_0$, the first four columns (128 bits) are set to sk_0, and the remaining two columns (64 bits) are set to the left half of sk_1. Among the 192-bit of the $Kstate_1$, the first two columns (64 bits) are set to the right half of sk_1, and the remaining four columns (128 bits) are set to sk_2. Similarly, $sk_3, sk_4, \ldots, sk_{12}$ are obtained.

Note that sk_{11} is the last four columns of $Kstate_7$, and then sk_{12} is the first four columns of $Kstate_8$. The last two columns of $Kstate_8$ are never used. Thus, in order to omit the redundant computations, the KSF should be processed up to the first four columns of $Kstate_8$.

The key schedule procedure for AES-192 is depicted in Figure 1.10.

1.5.3.2 The Key Schedule Function for AES-256

The KSF for AES-256 can be similarly defined. The size of the key state is 256 bits consisting of 4×8-byte array. Each key state produces two subkeys, and 15 subkeys $sk_0, sk_1, \ldots, sk_{14}$ are generated.

The update computation is very similar to the ones for AES-128 and AES-192. In order to mix the data quickly, another S-box layer is introduced between columns 3 and 4. The detailed procedure is omitted. The key schedule procedure for AES-256 is depicted in Figure 1.11.

Note that sk_{14} is the first four columns of $Kstate_7$. The last four columns of $Kstate_7$ are never used. Thus, in order to omit the redundant computations, the KSF should be processed up to the first four columns of $Kstate_7$.

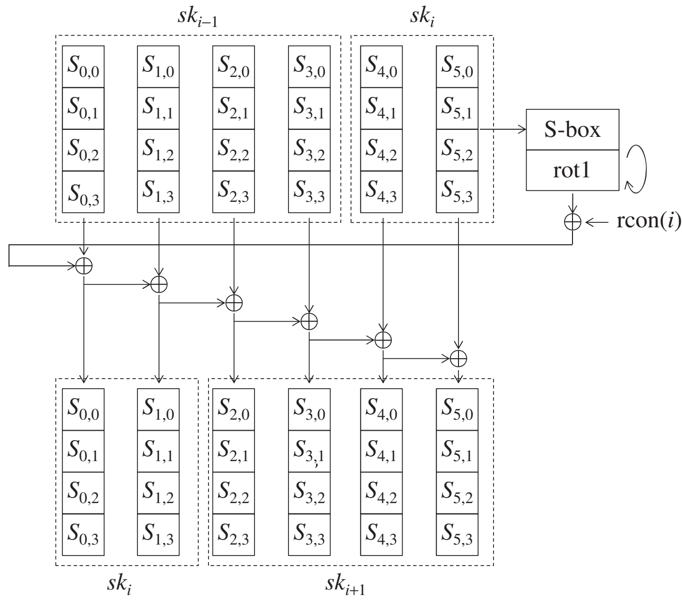

Figure 1.10 Key schedule function of AES-192. The key schedule function is iterated until 13 subkeys are generated

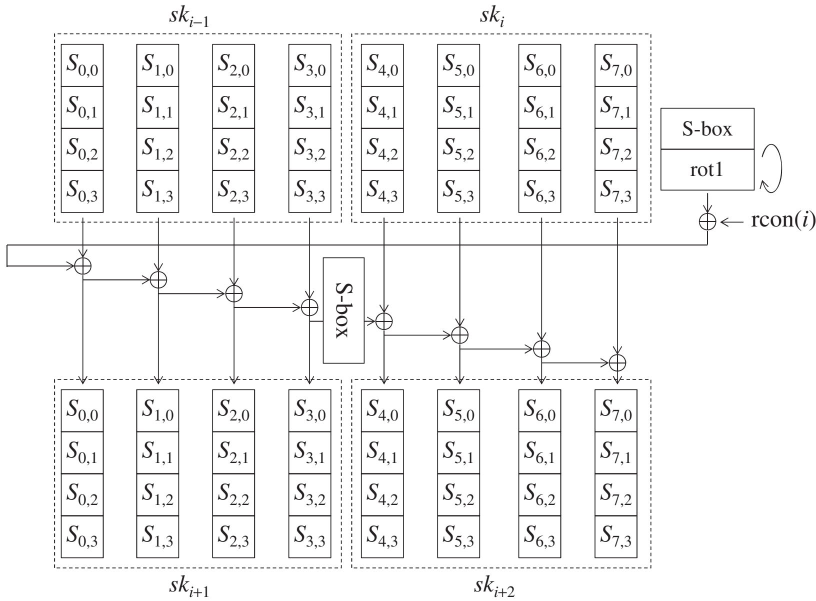

Figure 1.11 Key schedule function of AES-256. The key schedule function is iterated until 15 subkeys are generated

Exercise 1.9 *Compare the number of KSF calls per 128-bit subkeys for AES-128, AES-192, and AES-256 (weak mixing effect of the AES-256 KSF).*

Exercise 1.10 *Compare the number of S-box calls per 128-bit subkeys for AES-128, AES-192, and AES-256 (weak nonlinearity of the AES-192 KSF).*

1.5.4 Notations to Describe AES-128

The computation of AES-128 with all the operations is described in Figure 1.12. The state after the first XOR of the plaintext and sk_0 is denoted by S_1^I. Similarly in round i, where $i \in \{1, 2, \ldots, 10\}$,

- the state at the beginning of the round is denoted by S_i^I;
- the state after the $\mathrm{SubBytes}$ operation is denoted by S_i^{SB};

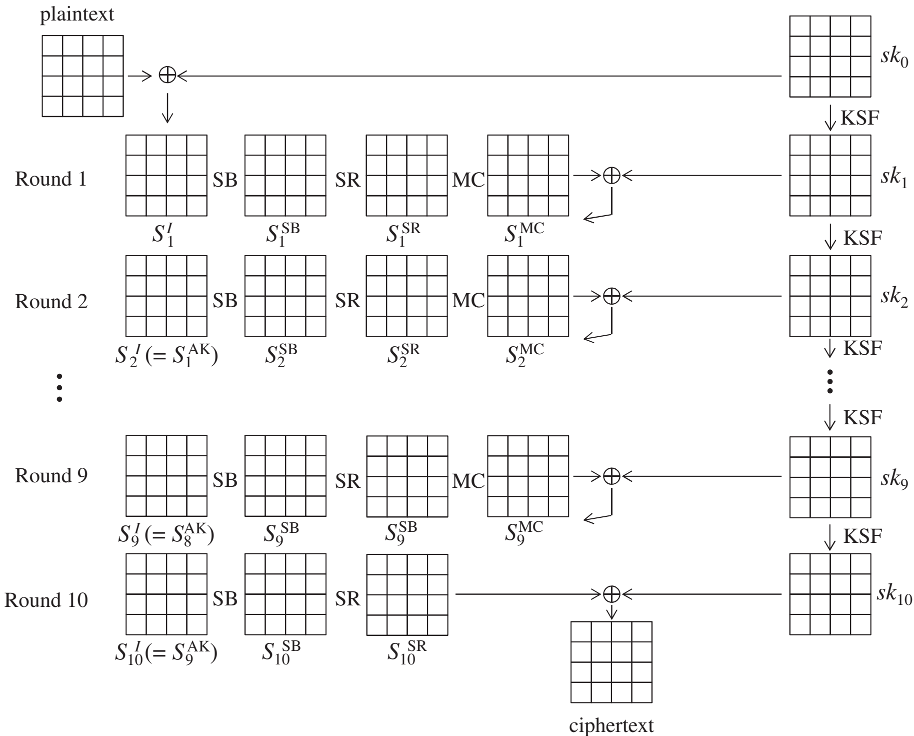

Figure 1.12 Notations for each state of AES-128

Inverse diagonal ↘ Diagonal ↙

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Figure 1.13 Notations for inside AES state

- the state after the ShiftRows operation is denoted by S_i^{SR};
- the state after the MixColumns operation is denoted by S_i^{MC};
- the state after the AddRoundKey operation is denoted by S_i^{AK}, which is equivalent to S_{i+1}^{I}.

As explained before, the state is represented by a 4×4-byte array. Using two subindices often causes misunderstanding, and thus each byte position is also denoted by a single sequence $\{0, 1, \ldots, 15\}$. For state S, the byte $S_{u,v}$, where $0 \leq u, v \leq 3$ is converted to the byte $S[4 * u + v]$. The byte positions in the single sequence are shown in Figure 1.13.

Byte values of state S in several different byte positions $[a], [b], [c], \ldots$ are often denoted by $S[a, b, c, \ldots]$. For example, the 4-byte value in the column 0 of state S is denoted by $S[0, 1, 2, 3]$.

- The first column, or column 0, of state S is denoted by $S[Col(0)]$, which is equivalent to $S[0, 1, 2, 3]$.
- The second column, or column 1, of state S is denoted by $S[Col(1)]$, which is equivalent to $S[4, 5, 6, 7]$.
- The third column, or column 2, of state S is denoted by $S[Col(2)]$, which is equivalent to $S[8, 9, 10, 11]$.
- The fourth column, or column 3, of state S is denoted by $S[Col(3)]$, which is equivalent to $S[12, 13, 14, 15]$.
- The first row, or row 0, of state S is denoted by $S[Row(0)]$, which is equivalent to $S[0, 4, 8, 12]$.
- The second row, or row 1, of state S is denoted by $S[Row(1)]$, which is equivalent to $S[1, 5, 9, 13]$.
- The third row, or row 2, of state S is denoted by $S[Row(2)]$, which is equivalent to $S[2, 6, 10, 14]$.
- The fourth row, or row 3, of state S is denoted by $S[Row(3)]$, which is equivalent to $S[3, 7, 11, 15]$.

State S_i^{SB} becomes state S_i^{SR} after the ShiftRows operation. During this process, 4 bytes in $S_i^{\mathrm{SB}}[Col(j)]$ are moved to different byte positions in S_i^{SR}. The moved positions are denoted by $\mathrm{SR}(Col(j))$.

- For state S, 4 bytes of $S[\mathrm{SR}(Col(0))]$ are equivalent to $S[0, 7, 10, 13]$.
- For state S, 4 bytes of $S[\mathrm{SR}(Col(1))]$ are equivalent to $S[1, 4, 11, 14]$.
- For state S, 4 bytes of $S[\mathrm{SR}(Col(2))]$ are equivalent to $S[2, 5, 8, 15]$.
- For state S, 4 bytes of $S[\mathrm{SR}(Col(3))]$ are equivalent to $S[3, 6, 9, 12]$.

Those 4-byte positions are called **diagonal**.

Similarly 4 bytes in $S_i^{\mathrm{SR}}[Col(j)]$ are moved to different byte positions in S_i^{SB} through the inverse of the $\mathrm{ShiftRows}$ operation. The moved positions are denoted by $\mathrm{SR}^{-1}(Col(j))$.

- For state S, 4 bytes of $S[\mathrm{SR}^{-1}(Col(0))]$ are equivalent to $S[0, 5, 10, 15]$.
- For state S, 4 bytes of $S[\mathrm{SR}^{-1}(Col(1))]$ are equivalent to $S[3, 4, 9, 14]$.
- For state S, 4 bytes of $S[\mathrm{SR}^{-1}(Col(2))]$ are equivalent to $S[2, 7, 8, 13]$.
- For state S, 4 bytes of $S[\mathrm{SR}^{-1}(Col(3))]$ are equivalent to $S[1, 6, 11, 12]$.

Those 4-byte positions are called **inverse diagonal**.

Further Reading

Daemen J and Rijmen V June 1998 *AES submission document on Rijndael*.

Daemen J and Rijmen V 2002 *The Design of Rijndeal: AES—The Advanced Encryption Standard (AES)*. Springer-Verlag.

Deschamps JP 2009 *Hardware Implementation of Finite-Field Arithmetic* 1st edn. McGraw-Hill, Inc., New York, NY.

Paar C and Pelzl J 2010 *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer-Verlag, New York.