Chapter 4
The Advanced Encryption Standard (AES)

The *Advanced Encryption Standard (AES)* is the most widely used symmetric cipher today. Even though the term "Standard" in its name only refers to US government applications, the AES block cipher is also mandatory in several industry standards and is used in many commercial systems. Among the commercial standards that include AES are the Internet security standard IPsec, TLS, the Wi-Fi encryption standard IEEE 802.11i, the secure shell network protocol SSH (Secure Shell), the Internet phone Skype and numerous security products around the world. To date, there are no attacks better than brute-force known against AES.

In this chapter you will learn:

- The design process of the US symmetric encryption standard, AES
- The encryption and decryption function of AES
- The internal structure of AES, namely:
  - byte substitution layer
  - diffusion layer
  - key addition layer
  - key schedule
- Basic facts about Galois fields
- Efficiency of AES implementations

4.1 Introduction

In 1999 the US National Institute of Standards and Technology (NIST) indicated that DES should only be used for legacy systems and instead triple DES (3DES) should be used. Even though 3DES resists brute-force attacks with today's technology, there are several problems with it. First, it is not very efficient with regard to software implementations. DES is already not particularly well suited for software and 3DES is three times slower than DES. Another disadvantage is the relatively short block size of 64 bits, which is a drawback in certain applications, e.g., if one wants to built a hash function from a block cipher (cf. Sect. 11.3.2). Finally, if one is worried about attacks with quantum computers, which might become reality in a few decades, key lengths on the order of 256 bits are desirable. All these consideration led NIST to the conclusion that an entirely new block cipher was needed as a replacement for DES.

In 1997 NIST called for proposals for a new *Advanced Encryption Standard (AES)*. Unlike the DES development, the selection of the algorithm for AES was an open process administered by NIST. In three subsequent AES evaluation rounds, NIST and the international scientific community discussed the advantages and disadvantages of the submitted ciphers and narrowed down the number of potential candidates. In 2001, NIST declared the block cipher *Rijndael* as the new AES and published it as a final standard (FIPS PUB 197). Rijndael was designed by two young Belgian cryptographers.

Within the call for proposals, the following requirements for all AES candidate submissions were mandatory:

- block cipher with 128 bit block size
- three key lengths must be supported: 128, 192 and 256 bit
- security relative to other submitted algorithms
- efficiency in software and hardware

The invitation for submitting suitable algorithms and the subsequent evaluation of the successor of DES was a public process. A compact chronology of the AES selection process is given here:

- The need for a new block cipher was announced on January 2, 1997, by NIST.
- A formal call for AES was announced on September 12, 1997.
- Fifteen candidate algorithms were submitted by researchers from several countries by August 20, 1998.
- On August 9, 1999, five finalist algorithms were announced:
    - *Mars* by IBM Corporation
    - *RC6* by RSA Laboratories
    - *Rijndael*, by Joan Daemen and Vincent Rijmen
    - *Serpent*, by Ross Anderson, Eli Biham and Lars Knudsen
    - *Twofish*, by Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall and Niels Ferguson

- On October 2, 2000, NIST announced that it had chosen Rijndael as the AES.
- On November 26, 2001, AES was formally approved as a US federal standard.

It is expected that AES will be the dominant symmetric-key algorithm for many commercial applications for the next few decades. It is also remarkable that in 2003 the US National Security Agency (NSA) announced that it allows AES to encrypt classified documents up to the level SECRET for all key lengths, and up to the TOP SECRET level for key lengths of either 192 or 256 bits. Prior to that date, only non-public algorithms had been used for the encryption of classified documents.

4.2 Overview of the AES Algorithm

The AES cipher is almost identical to the block cipher Rijndael. The Rijndael block and key size vary between 128, 192 and 256 bits. However, the AES standard only calls for a block size of 128 bits. Hence, only Rijndael with a block length of 128 bits is known as the AES algorithm. In the remainder of this chapter, we only discuss the standard version of Rijndael with a block length of 128 bits.

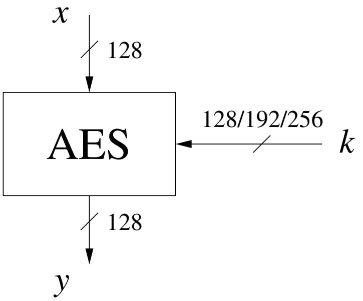

Fig. 4.1 AES input/output parameters

As mentioned previously, three key lengths must be supported by Rijndael as this was an NIST design requirement. The number of internal rounds of the cipher is a function of the key length according to Table 4.1.

Table 4.1 Key lengths and number of rounds for AES

key lengths	# rounds = n_r
128 bit	10
192 bit	12
256 bit	14

In contrast to DES, AES does not have a Feistel structure. Feistel networks do not encrypt an entire block per iteration, e.g., in DES, $64/2 = 32$ bits are encrypted in one round. AES, on the other hand, encrypts all 128 bits in one iteration. This is one reason why it has a comparably small number of rounds.

AES consists of so-called *layers*. Each layer manipulates all 128 bits of the data path. The data path is also referred to as the state of the algorithm. There are only three different types of layers. Each round, with the exception of the first, consists of all three layers as shown in Fig. 4.2: the plaintext is denoted as x, the ciphertext as y and the number of rounds as n_r. Moreover, the last round n_r does not make use of the MixColumn transformation, which makes the encryption and decryption scheme symmetric.

We continue with a brief description of the layers:

Key Addition layer A 128-bit round key, or subkey, which has been derived from the main key in the key schedule, is XORed to the state.

Byte Substitution layer (S-Box) Each element of the state is nonlinearly transformed using lookup tables with special mathematical properties. This introduces *confusion* to the data, i.e., it assures that changes in individual state bits propagate quickly across the data path.

Diffusion layer It provides *diffusion* over all state bits. It consists of two sublayers, both of which perform linear operations:

- The *ShiftRows* layer permutes the data on a byte level.
- The *MixColumn* layer is a matrix operation which combines (mixes) blocks of four bytes.

Similar to DES, the key schedule computes round keys, or subkeys, $(k_0, k_1, \ldots, k_{n_r})$ from the original AES key.

Before we describe the internal functions of the layers in Sect. 4.4, we have to introduce a new mathematical concept, namely *Galois fields*. Galois field computations are needed for all operations within the AES layers.

4.3 Some Mathematics: A Brief Introduction to Galois Fields

In AES, Galois field arithmetic is used in most layers, especially in the S-Box and the MixColumn layer. Hence, for a deeper understanding of the internals of AES, we provide an introduction to Galois fields as needed for this purpose before we continue with the algorithm in Sect. 4.4. A background on Galois fields is not required for a basic understanding of AES, and the reader can skip this section.

4.3.1 Existence of Finite Fields

A *finite field*, sometimes also called *Galois field*, is a set with a finite number of elements. Roughly speaking, a Galois field is a finite set of elements in which we

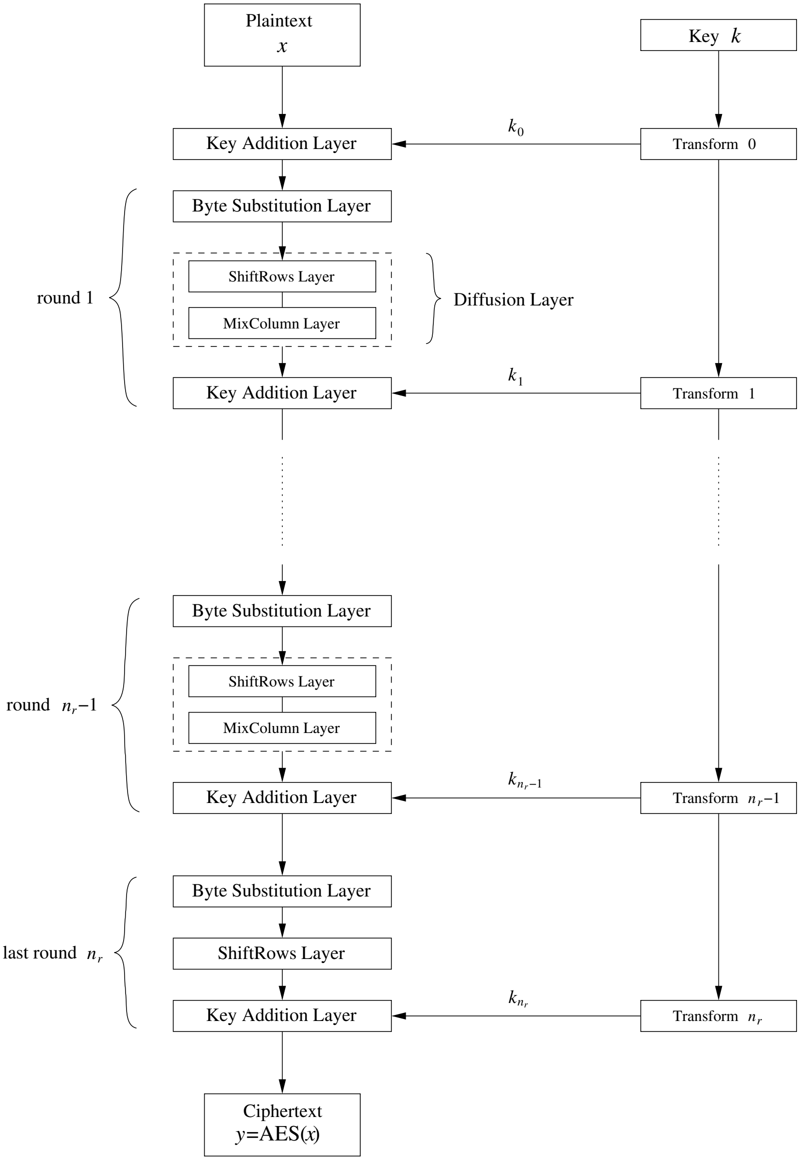

Fig. 4.2 AES encryption block diagram

can add, subtract, multiply and invert. Before we introduce the definition of a field, we first need the concept of a a simpler algebraic structure, a group.

Definition 4.3.1 Group
A group *is a set of elements G together with an operation* $\circ$ *which combines two elements of G. A group has the following properties:*

1. *The group operation* $\circ$ *is* closed. *That is, for all* $a, b, \in G$*, it holds that* $a \circ b = c \in G$*.*
2. *The group operation is* associative. *That is,* $a \circ (b \circ c) = (a \circ b) \circ c$ *for all* $a, b, c \in G$*.*
3. *There is an element* $1 \in G$*, called the* neutral element *(or* identity element*), such that* $a \circ 1 = 1 \circ a = a$ *for all* $a \in G$*.*
4. *For each* $a \in G$ *there exists an element* $a^{-1} \in G$*, called the* inverse *of a, such that* $a \circ a^{-1} = a^{-1} \circ a = 1$.
5. *A group G is* abelian (or commutative) *if, furthermore,* $a \circ b = b \circ a$ *for all* $a, b \in G$*.*

Roughly speaking, a group is set with one operation and the corresponding inverse operation. If the operation is called addition, the inverse operation is subtraction; if the operation is multiplication, the inverse operation is division (or multiplication with the inverse element).

Example 4.1. The set of integers $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$ and the operation addition modulo m form a group with the neutral element 0. Every element a has an inverse $-a$ such that $a + (-a) = 0 \bmod m$. Note that this set does not form a group with the operation multiplication because most elements a do not have an inverse such that $aa^{-1} = 1 \bmod m$.
◇

In order to have all four basic arithmetic operations (i.e., addition, subtraction, multiplication, division) in one structure, we need a set which contains an additive and a multiplicative group. This is what we call a field.

Definition 4.3.2 Field
A field *F is a set of elements with the following properties:*

- *All elements of F form an additive group with the group operation "+" and the neutral element 0.*
- *All elements of F except 0 form a multiplicative group with the group operation "×" and the neutral element 1.*
- *When the two group operations are mixed, the distributivity law holds, i.e., for all* $a, b, c \in F$*:* $a(b + c) = (ab) + (ac)$*.*

Example 4.2. The set $\mathbb{R}$ of real numbers is a field with the neutral element 0 for the additive group and the neutral element 1 for the multiplicative group. Every real number a has an additive inverse, namely $-a$, and every nonzero element a has a multiplicative inverse $1/a$.

◇

In cryptography, we are almost always interested in fields with a finite number of elements, which we call finite fields or Galois fields. The number of elements in the field is called the *order* or *cardinality* of the field. Of fundamental importance is the following theorem:

> **Theorem 4.3.1** *A field with order m only exists if m is a prime power, i.e., $m = p^n$, for some positive integer n and prime integer p. p is called the* characteristic *of the finite field.*

This theorem implies that there are, for instance, finite fields with 11 elements, or with 81 elements (since $81 = 3^4$) or with 256 elements (since $256 = 2^8$, and 2 is a prime). However, there is no finite field with 12 elements since $12 = 2^2 \cdot 3$, and 12 is thus not a prime power. In the remainder of this section we look at how finite fields can be built, and more importantly for our purpose, how we can do arithmetic in them.

4.3.2 Prime Fields

The most intuitive examples of finite fields are fields of prime order, i.e., fields with $n = 1$. Elements of the field $GF(p)$ can be represented by integers $0, 1, \ldots, p-1$. The two operations of the field are modular integer addition and integer multiplication *modulo p*.

> **Theorem 4.3.2** *Let p be a prime. The integer ring $\mathbb{Z}_p$ is denoted as $GF(p)$ and is referred to as a* prime field*, or as a* Galois field *with a prime number of elements. All nonzero elements of $GF(p)$ have an inverse. Arithmetic in $GF(p)$ is done modulo p.*

This means that if we consider the integer ring $\mathbb{Z}_m$ which was introduced in Sect. 1.4.2, i.e., integers with modular addition and multiplication, and m happens to be a prime, $\mathbb{Z}_m$ is not only a ring but also a finite field.

In order to do arithmetic in a prime field, we have to follow the rules for integer rings: Addition and multiplication are done modulo p, the additive inverse of any element a is given by $a + (-a) = 0 \bmod p$, and the multiplicative inverse of any nonzero element a is defined as $a \cdot a^{-1} = 1$. Let's have a look at an example of a prime field.

Example 4.3. We consider the finite field $GF(5) = \{0, 1, 2, 3, 4\}$. The tables below describe how to add and multiply any two elements, as well as the additive and

addition

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

additive inverse

$-0 = 0$
$-1 = 4$
$-2 = 3$
$-3 = 2$
$-4 = 1$

multiplication

×	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

multiplicative inverse

0^{-1} does not exist
$1^{-1} = 1$
$2^{-1} = 3$
$3^{-1} = 2$
$4^{-1} = 4$

multiplicative inverse of the field elements. Using these tables, we can perform all calculations in this field without using modular reduction explicitly.

◇

A very important prime field is $GF(2)$, which is the smallest finite field that exists. Let's have a look at the multiplication and addition tables for the field.

Example 4.4. Let's consider the small finite field $GF(2) = \{0,1\}$. Arithmetic is simply done modulo 2, yielding the following arithmetic tables:

addition

+	0	1
0	0	1
1	1	0

multiplication

×	0	1
0	0	0
1	0	1

◇

As we saw in Chap. 2 on stream ciphers, $GF(2)$ addition, i.e., modulo 2 addition, is equivalent to an XOR gate. What we learn from the example above is that $GF(2)$ multiplication is equivalent to the logical AND gate. The field $GF(2)$ is important for AES.

4.3.3 Extension Fields $GF(2^m)$

In AES the finite field contains 256 elements and is denoted as $GF(2^8)$. This field was chosen because each of the field elements can be represented by one byte. For the S-Box and MixColumn transforms, AES treats every byte of the internal data path as an element of the field $GF(2^8)$ and manipulates the data by performing arithmetic in this finite field.

However, if the order of a finite field is not prime, and 2^8 is clearly not a prime, the addition and multiplication operation cannot be represented by addition and multiplication of integers modulo 2^8. Such fields with $m > 1$ are called *extension fields*. In order to deal with extension fields we need (1) a different notation for field elements and (2) different rules for performing arithmetic with the elements. We will see in the following that elements of extension fields can be represented as *polynomials*, and that computation in the extension field is achieved by performing a certain type of *polynomial arithmetic*.

In extension fields $GF(2^m)$ elements are not represented as integers but as polynomials with coefficients in $GF(2)$. The polynomials have a maximum degree of $m-1$, so that there are m coefficients in total for every element. In the field $GF(2^8)$, which is used in AES, each element $A \in GF(2^8)$ is thus represented as:

$$A(x) = a_7x^7 + \cdots + a_1x + a_0, \quad a_i \in GF(2) = \{0,1\}.$$

Note that there are exactly $256 = 2^8$ such polynomials. The set of these 256 polynomials is the finite field $GF(2^8)$. It is also important to observe that every polynomial can simply be stored in digital form as an 8-bit vector

$$A = (a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0).$$

In particular, we do *not* have to store the factors x^7, x^6, etc. It is clear from the bit positions to which power x^i each coefficient belongs.

4.3.4 Addition and Subtraction in $GF(2^m)$

Let's now look at addition and subtraction in extension fields. The key addition layer of AES uses addition. It turns out that these operations are straightforward. They are simply achieved by performing standard polynomial addition and subtraction: We merely add or subtract coefficients with equal powers of x. The coefficient additions or subtractions are done in the underlying field $GF(2)$.

Definition 4.3.3 Extension field addition and subtraction
Let $A(x), B(x) \in GF(2^m)$. The sum of the two elements is then computed according to:

$$C(x) = A(x) + B(x) = \sum_{i=0}^{m-1} c_i x^i, \quad c_i \equiv a_i + b_i \bmod 2$$

and the difference is computed according to:

$$C(x) = A(x) - B(x) = \sum_{i=0}^{m-1} c_i x^i, \quad c_i \equiv a_i - b_i \equiv a_i + b_i \bmod 2.$$

Note that we perform modulo 2 addition (or subtraction) with the coefficients. As we saw in Chap. 2, addition and subtraction modulo 2 are the same operation. Moreover, addition modulo 2 is equal to bitwise XOR. Let's have a look at an example in the field $GF(2^8)$ which is used in AES:

Example 4.5. Here is how the sum $C(x) = A(x) + B(x)$ of two elements from $GF(2^8)$ is computed:

$$\begin{array}{lllllll} A(x) & = x^7 + & x^6 + & x^4 + & & & 1 \\ B(x) & = & & x^4 + & x^2 + & & 1 \\ \hline C(x) & = x^7 + & x^6 + & & x^2 & & \end{array}$$

◇

Note that if we computed the difference of the two polynomials $A(x) - B(x)$ from the example above, we would get the same result as for the sum.

4.3.5 Multiplication in $GF(2^m)$

Multiplication in $GF(2^8)$ is the core operation of the MixColumn transformation of AES. In a first step, two elements (represented by their polynomials) of a finite field $GF(2^m)$ are multiplied using the standard polynomial multiplication rule:

$$\begin{aligned} A(x) \cdot B(x) &= (a_{m-1}x^{m-1} + \cdots + a_0) \cdot (b_{m-1}x^{m-1} + \cdots + b_0) \\ C'(x) &= c'_{2m-2}x^{2m-2} + \cdots + c'_0, \end{aligned}$$

where:

$$\begin{aligned} c'_0 &= a_0 b_0 \bmod 2 \\ c'_1 &= a_0 b_1 + a_1 b_0 \bmod 2 \\ &\vdots \\ c'_{2m-2} &= a_{m-1} b_{m-1} \bmod 2. \end{aligned}$$

Note that all coefficients a_i, b_i and c_i are elements of $GF(2)$, and that coefficient arithmetic is performed in $GF(2)$. In general, the product polynomial $C(x)$ will have a degree higher than $m-1$ and has to be reduced. The basic idea is an approach similar to the case of multiplication in prime fields: in $GF(p)$, we multiply the two integers, divide the result by a prime, and consider only the remainder. Here is what we are doing in extension fields: The product of the multiplication is divided by a certain polynomial, and we consider only the remainder after the polynomial division. We need irreducible polynomials for the module reduction. We recall from Sect. 2.3.1 that irreducible polynomials are roughly comparable to prime numbers, i.e., their only factors are 1 and the polynomial itself.

Definition 4.3.4 Extension field multiplication
Let $A(x), B(x) \in GF(2^m)$ *and let*

$$P(x) \equiv \sum_{i=0}^{m} p_i x^i, \quad p_i \in GF(2)$$

be an irreducible polynomial. Multiplication of the two elements $A(x), B(x)$ *is performed as*

$$C(x) \equiv A(x) \cdot B(x) \bmod P(x).$$

Thus, every field $GF(2^m)$ requires an irreducible polynomial $P(x)$ of degree m with coefficients from $GF(2)$. Note that not all polynomials are irreducible. For example, the polynomial x^4+x^3+x+1 is reducible since

$$x^4+x^3+x+1 = (x^2+x+1)(x^2+1)$$

and hence cannot be used to construct the extension field $GF(2^4)$. Since primitive polynomials are a special type of irreducible polynomial, the polynomials in Table 2.3 can be used for constructing fields $GF(2^m)$. For AES, the irreducible polynomial

$$P(x) = x^8+x^4+x^3+x+1$$

is used. It is part of the AES specification.

Example 4.6. We want to multiply the two polynomials $A(x) = x^3+x^2+1$ and $B(x) = x^2+x$ in the field $GF(2^4)$. The irreducible polynomial of this Galois field is given as

$$P(x) = x^4+x+1.$$

The plain polynomial product is computed as:

$$C'(x) = A(x) \cdot B(x) = x^5+x^3+x^2+x.$$

We can now reduce $C'(x)$ using the polynomial division method we learned in school. However, sometimes it is easier to reduce each of the leading terms x^4 and

x^5 individually:

$$\begin{aligned} x^4 &= 1 \cdot P(x) + (x+1) \\ x^4 &\equiv x+1 \bmod P(x) \\ x^5 &\equiv x^2 + x \bmod P(x). \end{aligned}$$

Now, we only have to insert the reduced expression for x^5 into the intermediate result $C'(x)$:

$$\begin{aligned} C(x) &\equiv x^5 + x^3 + x^2 + x \bmod P(x) \\ C(x) &\equiv (x^2 + x) + (x^3 + x^2 + x) = x^3 \\ A(x) \cdot B(x) &\equiv x^3. \end{aligned}$$

◇

It is important not to confuse multiplication in $GF(2^m)$ with integer multiplication, especially if we are concerned with software implementations of Galois fields. Recall that the polynomials, i.e., the field elements, are normally stored as bit vectors in the computers. If we look at the multiplication from the previous example, the following very atypical operation is being performed on the bit level:

$$\begin{array}{ccccc} A & \cdot & B & = & C \\ (x^3 + x^2 + 1) & \cdot & (x^2 + x) & = & x^3 \\ (1\,1\,0\,1) & \cdot & (0\,1\,1\,0) & = & (1\,0\,0\,0). \end{array}$$

This computation is **not** identical to integer arithmetic. If the polynomials are interpreted as integers, i.e., $(1101)_2 = 13_{10}$ and $(0110)_2 = 6_{10}$, the result would have been $(1001110)_2 = 78_{10}$, which is clearly not the same as the Galois field multiplication product. Hence, even though we can represent field elements as integers data types, we cannot make use of the integer arithmetic provided

4.3.6 Inversion in $GF(2^m)$

Inversion in $GF(2^8)$ is the core operation of the Byte Substitution transformation, which contains the AES S-Boxes. For a given finite field $GF(2^m)$ and the corresponding irreducible reduction polynomial $P(x)$, the inverse A^{-1} of a nonzero element $A \in GF(2^m)$ is defined as:

$$A^{-1}(x) \cdot A(x) = 1 \bmod P(x).$$

For small fields — in practice this often means fields with 2^{16} or fewer elements — lookup tables which contain the precomputed inverses of all field elements are often used. Table 4.2 shows the values which are used within the S-Box of AES. The table contains all inverses in $GF(2^8)$ modulo $P(x) = x^8 + x^4 + x^3 + x + 1$ in hexadecimal notation. A special case is the entry for the field element 0, for which an inverse does not exist. However, for the AES S-Box, a substitution table is needed that is defined for every possible input value. Hence, the designers defined the S-Box such that the input value 0 is mapped to the output value 0.

Table 4.2 Multiplicative inverse table in $GF(2^8)$ for bytes xy used within the AES S-Box

		Y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	00	01	8D	F6	CB	52	7B	D1	E8	4F	29	C0	B0	E1	E5	C7
	1	74	B4	AA	4B	99	2B	60	5F	58	3F	FD	CC	FF	40	EE	B2
	2	3A	6E	5A	F1	55	4D	A8	C9	C1	0A	98	15	30	44	A2	C2
	3	2C	45	92	6C	F3	39	66	42	F2	35	20	6F	77	BB	59	19
	4	1D	FE	37	67	2D	31	F5	69	A7	64	AB	13	54	25	E9	09
	5	ED	5C	05	CA	4C	24	87	BF	18	3E	22	F0	51	EC	61	17
	6	16	5E	AF	D3	49	A6	36	43	F4	47	91	DF	33	93	21	3B
	7	79	B7	97	85	10	B5	BA	3C	B6	70	D0	06	A1	FA	81	82
X	8	83	7E	7F	80	96	73	BE	56	9B	9E	95	D9	F7	02	B9	A4
	9	DE	6A	32	6D	D8	8A	84	72	2A	14	9F	88	F9	DC	89	9A
	A	FB	7C	2E	C3	8F	B8	65	48	26	C8	12	4A	CE	E7	D2	62
	B	0C	E0	1F	EF	11	75	78	71	A5	8E	76	3D	BD	BC	86	57
	C	0B	28	2F	A3	DA	D4	E4	0F	A9	27	53	04	1B	FC	AC	E6
	D	7A	07	AE	63	C5	DB	E2	EA	94	8B	C4	D5	9D	F8	90	6B
	E	B1	0D	D6	EB	C6	0E	CF	AD	08	4E	D7	E3	5D	50	1E	B3
	F	5B	23	38	34	68	46	03	8C	DD	9C	7D	A0	CD	1A	41	1C

Example 4.7. From Table 4.2 the inverse of

$$x^7+x^6+x=(11000010)_2=(C2)_{hex}=(xy)$$

is given by the element in row C, column 2:

$$(2F)_{hex}=(00101111)_2=x^5+x^3+x^2+x+1.$$

This can be verified by multiplication:

$$(x^7+x^6+x)\cdot(x^5+x^3+x^2+x+1)\equiv 1 \mod P(x).$$

◇

Note that the table above does not contain the S-Box itself, which is a bit more complex and will be described in Sect. 4.4.1.

As an alternative to using lookup tables, one can also explicitly compute inverses. The main algorithm for computing multiplicative inverses is the extended Euclidean algorithm, which is introduced in Sect. 6.3.1.

4.4 Internal Structure of AES

In the following, we examine the internal structure of AES. Figure 4.3 shows the graph of a single AES round. The 16-byte input $A_0,\dots,A_{15}$ is fed byte-wise into the

S-Box. The 16-byte output $B_0,\ldots,B_{15}$ is permuted byte-wise in the ShiftRows layer and mixed by the MixColumn transformation $c(x)$. Finally, the 128-bit subkey k_i is XORed with the intermediate result. We note that AES is a byte-oriented cipher.

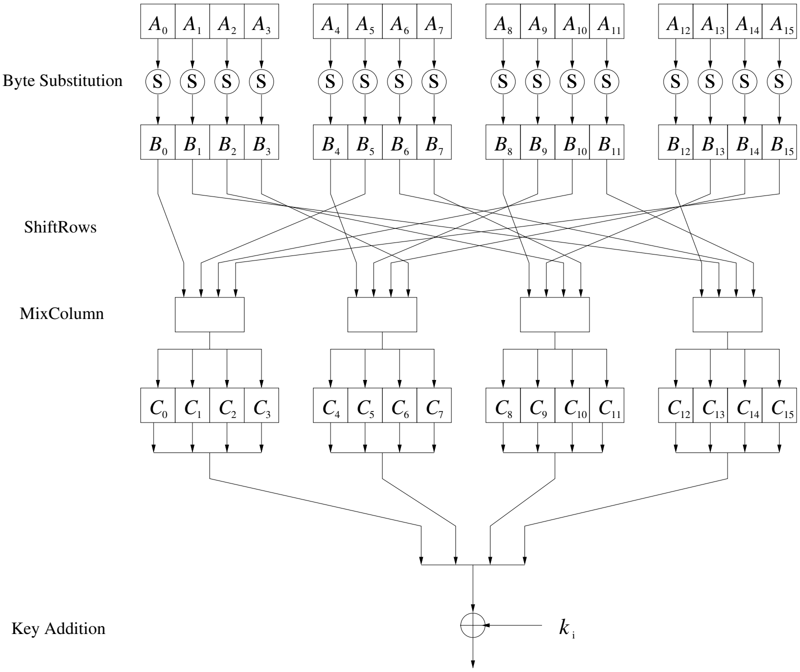
A_0 A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10} A_{11} A_{12} A_{13} A_{14} A_{15}
Byte Substitution
S
B_0 B_1 B_2 B_3 B_4 B_5 B_6 B_7 B_8 B_9 B_{10} B_{11} B_{12} B_{13} B_{14} B_{15}
ShiftRows
MixColumn
C_0 C_1 C_2 C_3 C_4 C_5 C_6 C_7 C_8 C_9 C_{10} C_{11} C_{12} C_{13} C_{14} C_{15}
Key Addition
k_i

Fig. 4.3 AES round function for rounds $1,2,\ldots,n_r-1$

This is in contrast to DES, which makes heavy use of bit permutation and can thus be considered to have a bit-oriented structure.

In order to understand how the data moves through AES, we first imagine that the state A (i.e., the 128-bit data path) consisting of 16 bytes $A_0,A_1,\ldots,A_{15}$ is arranged in a four-by-four byte matrix:

A_0	A_4	A_8	A_{12}
A_1	A_5	A_9	A_{13}
A_2	A_6	A_{10}	A_{14}
A_3	A_7	A_{11}	A_{15}

As we will see in the following, AES operates on elements, columns or rows of the current state matrix. Similarly, the key bytes are arranged into a matrix with four rows and four (128-bit key), six (192-bit key) or eight (256-bit key) columns. Here is, as an example, the state matrix of a 192-bit key:

K_0	K_4	K_8	K_{12}	K_{16}	K_{20}
K_1	K_5	K_9	K_{13}	K_{17}	K_{21}
K_2	K_6	K_{10}	K_{14}	K_{18}	K_{22}
K_3	K_7	K_{11}	K_{15}	K_{19}	K_{23}

We discuss now what happens in each of the layers.

4.4.1 Byte Substitution Layer

As shown in Fig. 4.3, the first layer in each round is the *Byte Substitution layer*. The Byte Substitution layer can be viewed as a row of 16 parallel S-Boxes, each with 8 input and output bits. Note that all 16 S-Boxes are identical, unlike DES where eight different S-Boxes are used. In the layer, each state byte A_i is replaced, i.e., substituted, by another byte B_i:

$$S(A_i) = B_i.$$

The S-Box is the only nonlinear element of AES, i.e., it holds that ByteSub$(A)+$ ByteSub$(B) \neq$ ByteSub$(A+B)$ for two states A and B. The S-Box substitution is a bijective mapping, i.e., each of the $2^8 = 256$ possible input elements is one-to-one mapped to one output element. This allows us to uniquely reverse the S-Box, which is needed for decryption. In software implementations the S-Box is usually realized as a 256-by-8 bit lookup table with fixed entries, as given in Table 4.3.

Table 4.3 AES S-Box: Substitution values in hexadecimal notation for input byte (xy)

									y							
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
	4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
	5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
	6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
	7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
x	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

Example 4.8. Let's assume the input byte to the S-Box is $A_i = (C2)_{hex}$, then the substituted value is

$$S((C2)_{hex}) = (25)_{hex}.$$

On a bit level — and remember, the only thing that is ultimate of interest in encryption is the manipulation of bits — this substitution can be described as:

$$S(11000010) = (00100101).$$

◇

Even though the S-Box is bijective, it does not have any fixed points, i.e., there aren't any input values A_i such that $S(A_i) = A_i$. Even the zero-input is not a fixed point: $S(00000000) = (01100011)$.

Example 4.9. Let's assume the input to the Byte Substitution layer is

$$(C2, C2, \ldots, C2)$$

in hexadecimal notation. The output state is then

$$(25, 25, \ldots, 25).$$

◇

Mathematical description of the S-Box For readers who are interested in how the S-Box entries are constructed, a more detailed description now follows. This description, however, is not necessary for a basic understanding of AES, and the remainder of this subsection can be skipped without problem. Unlike the DES S-Boxes, which are essentially random tables that fulfill certain properties, the AES S-Boxes have a strong algebraic structure. An AES S-Box can be viewed as a two-step mathematical transformation (Fig. 4.4).

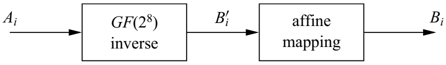

Fig. 4.4 The two operations within the AES S-Box which computes the function $B_i = S(A_i)$

The first part of the substitution is a Galois field inversion, the mathematics of which were introduced in Sect. 4.3.2. For each input element A_i, the inverse is computed: $B'_i = A_i^{-1}$, where both A_i and B'_i are considered elements in the field $GF(2^8)$ with the fixed irreducible polynomial $P(x) = x^8 + x^4 + x^3 + x + 1$. A lookup table with all inverses is shown in Table 4.2. Note that the inverse of the zero element does not exist. However, for AES it is defined that the zero element $A_i = 0$ is mapped to itself.

In the second part of the substitution, each byte B'_i is multiplied by a constant bit-matrix followed by the addition of a constant 8-bit vector. The operation is described by:

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} \equiv \begin{pmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{pmatrix} \begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \bmod 2.$$

Note that $B' = (b'_7, \ldots, b'_0)$ is the bitwise vector representation of $B'_i(x) = A_i^{-1}(x)$. This second step is referred to as *affine mapping*. Let's look at an example of how the S-Box computations work.

Example 4.10. We assume the S-Box input $A_i = (11000010)_2 = (C2)_{hex}$. From Table 4.2 we can see that the inverse is:

$$A_i^{-1} = B'_i = (2F)_{hex} = (00101111)_2.$$

We now apply the B'_i bit vector as input to the affine transformation. Note that the least significant bit (lsb) b'_0 of B'_i is at the rightmost position.

$$B_i = (00100101) = (25)_{hex}$$

Thus, $S((C2)_{hex}) = (25)_{hex}$, which is exactly the result that is also given in the S-Box Table 4.3.

◇

If one computes both steps for all 256 possible input elements of the S-Box and stores the results, one obtains Table 4.3. In most AES implementations, in particular in virtually all software realizations of AES, the S-Box outputs are *not explicitly computed* as shown here, but rather lookup tables like Table 4.3 are used. However, for hardware implementations it is sometimes advantageous to realize the S-Boxes as digital circuits which actually compute the inverse followed by the affine mapping.

The advantage of using inversion in $GF(2^8)$ as the core function of the Byte Substitution layer is that it provides a high degree of nonlinearity, which in turn provides optimum protection against some of the strongest known analytical attacks. The affine step "destroys" the algebraic structure of the Galois field, which in turn is needed to prevent attacks that would exploit the finite field inversion.

4.4.2 Diffusion Layer

In AES, the Diffusion layer consists of two sublayers, the *ShiftRows* transformation and the *MixColumn* transformation. We recall that diffusion is the spreading of the influence of individual bits over the entire state. Unlike the nonlinear S-Box, the diffusion layer performs a linear operation on state matrices A,B, i.e., $\text{DIFF}(A) + \text{DIFF}(B) = \text{DIFF}(A+B)$.

ShiftRows Sublayer

The ShiftRows transformation cyclically shifts the second row of the state matrix by three bytes to the right, the third row by two bytes to the right and the fourth row by one byte to the right. The first row is not changed by the ShiftRows transformation. The purpose of the ShiftRows transformation is to increase the diffusion properties of AES. If the input of the ShiftRows sublayer is given as a state matrix $B = (B_0, B_1, \ldots, B_{15})$:

B_0	B_4	B_8	B_{12}
B_1	B_5	B_9	B_{13}
B_2	B_6	B_{10}	B_{14}
B_3	B_7	B_{11}	B_{15}

the output is the new state:

$$\begin{array}{|c|c|c|c|l}
\hline
B_0 & B_4 & B_8 & B_{12} & \quad \text{no shift} \\
\hline
B_5 & B_9 & B_{13} & B_1 & \longrightarrow \text{three positions right shift} \\
\hline
B_{10} & B_{14} & B_2 & B_6 & \longrightarrow \text{two positions right shift} \\
\hline
B_{15} & B_3 & B_7 & B_{11} & \longrightarrow \text{one position right shift} \\
\hline
\end{array} \tag{4.1}$$

MixColumn Sublayer

The *MixColumn* step is a linear transformation which mixes each column of the state matrix. Since every input byte influences four output bytes, the MixColumn operation is the major diffusion element in AES. The combination of the ShiftRows and MixColumn layer makes it possible that after only three rounds every byte of the state matrix depends on all 16 plaintext bytes.

In the following, we denote the 16-byte input state by B and the 16-byte output state by C:

$$\text{MixColumn}(B) = C,$$

where B is the state after the ShiftRows operation as given in Expression (4.1).

Now, each 4-byte column is considered as a vector and multiplied by a fixed 4×4 matrix. The matrix contains *constant* entries. Multiplication and addition of the coefficients is done in $GF(2^8)$. As an example, we show how the first four output bytes are computed:

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} B_0 \\ B_5 \\ B_{10} \\ B_{15} \end{pmatrix}.$$

The second column of output bytes (C_4, C_5, C_6, C_7) is computed by multiplying the four input bytes (B_4, B_9, B_{14}, B_3) by the same constant matrix, and so on. Figure 4.3 shows which input bytes are used in each of the four MixColumn operations.

We discuss now the details of the vector–matrix multiplication which forms the MixColum operations. We recall that each state byte C_i and B_i is an 8-bit value representing an element from $GF(2^8)$. All arithmetic involving the coefficients is done in this Galois field. For the constants in the matrix a hexadecimal notation is used: "01" refers to the $GF(2^8)$ polynomial with the coefficients $(0000\,0001)$, i.e., it is the element 1 of the Galois field; "02" refers to the polynomial with the bit vector $(0000\,0010)$, i.e., to the polynomial x; and "03" refers to the polynomial with the bit vector $(0000\,0011)$, i.e., the Galois field element $x+1$.

The additions in the vector–matrix multiplication are $GF(2^8)$ additions, that is simple bitwise XORs of the respective bytes. For the multiplication of the constants, we have to realize multiplications with the constants 01, 02 and 03. These are quite efficient, and in fact, the three constants were chosen such that software implementation is easy. Multiplication by 01 is multiplication by the identity and does not involve any explicit operation. Multiplication by 02 and 03 can be done through table look-up in two 256-by-8 tables. As an alternative, multiplication by 02 can also be implemented as a multiplication by x, which is a left shift by one bit, and a modular reduction with $P(x) = x^8 + x^4 + x^3 + x + 1$. Similarly, multiplication by 03, which represents the polynomial $(x+1)$, can be implemented by a left shift by one bit and addition of the original value followed by a modular reduction with $P(x)$.

Example 4.11. We continue with our example from Sect. 4.4.1 and assume that the input state to the MixColumn layer is

$$B = (25, 25, \ldots, 25).$$

In this special case, only two multiplications in $GF(2^8)$ have to be done. These are $02 \cdot 25$ and $03 \cdot 25$, which can be computed in polynomial notation:

$$\begin{aligned} 02 \cdot 25 &= x \cdot (x^5 + x^2 + 1) \\ &= x^6 + x^3 + x, \\ 03 \cdot 25 &= (x+1) \cdot (x^5 + x^2 + 1) \\ &= (x^6 + x^3 + x) + (x^5 + x^2 + 1) \\ &= x^6 + x^5 + x^3 + x^2 + x + 1. \end{aligned}$$

Since both intermediate values have a degree smaller than 8, no modular reduction with $P(x)$ is necessary.

The output bytes of C result from the following addition in $GF(2^8)$:

$$\begin{array}{rcl}
01 \cdot 25 & = & \phantom{x^6+{}} x^5 + \phantom{x^3+{}} x^2 + \phantom{x+{}} 1 \\
01 \cdot 25 & = & \phantom{x^6+{}} x^5 + \phantom{x^3+{}} x^2 + \phantom{x+{}} 1 \\
02 \cdot 25 & = & x^6 + \phantom{x^5+{}} x^3 + \phantom{x^2+{}} x \\
03 \cdot 25 & = & x^6 + x^5 + x^3 + x^2 + x + 1 \\
\hline
C_i & = & \phantom{x^6+{}} x^5 + \phantom{x^3+{}} x^2 + \phantom{x+{}} 1,
\end{array}$$

where $i = 0, \ldots, 15$. This leads to the output state $C = (25, 25, \ldots, 25)$.

◇

4.4.3 Key Addition Layer

The two inputs to the *Key Addition layer* are the current 16-byte state matrix and a subkey which also consists of 16 bytes (128 bits). The two inputs are combined through a bitwise XOR operation. Note that the XOR operation is equal to addition in the Galois field $GF(2)$. The subkeys are derived in the key schedule that is described below in Sect. 4.4.4.

4.4.4 Key Schedule

The *key schedule* takes the original input key (of length 128, 192 or 256 bit) and derives the subkeys used in AES. Note that an XOR addition of a subkey is used both at the input and output of AES. This process is sometimes referred to as key whitening. The number of subkeys is equal to the number of rounds plus one, due to the key needed for key whitening in the first key addition layer, cf. Fig. 4.2. Thus, for the key length of 128 bits, the number of rounds is $n_r = 10$, and there are 11 subkeys, each of 128 bits. The AES with a 192-bit key requires 13 subkeys of length 128 bits, and AES with a 256-bit key has 15 subkeys. The AES subkeys are computed recursively, i.e., in order to derive subkey k_i, subkey k_{i-1} must be known, etc.

The AES key schedule is word-oriented, where 1 word = 32 bits. Subkeys are stored in a key expansion array W that consists of words. There are different key schedules for the three different AES key sizes of 128, 192 and 256 bit, which are all fairly similar. We introduce the three key schedules in the following.

Key Schedule for 128-Bit Key AES

The 11 subkeys are stored in a key expansion array with the elements $W[0], \ldots, W[43]$. The subkeys are computed as depicted in Fig. 4.5. The elements $K_0, \ldots, K_{15}$ denote the bytes of the original AES key.

First, we note that the first subkey k_0 is the original AES key, i.e., the key is copied into the first four elements of the key array W. The other array elements are

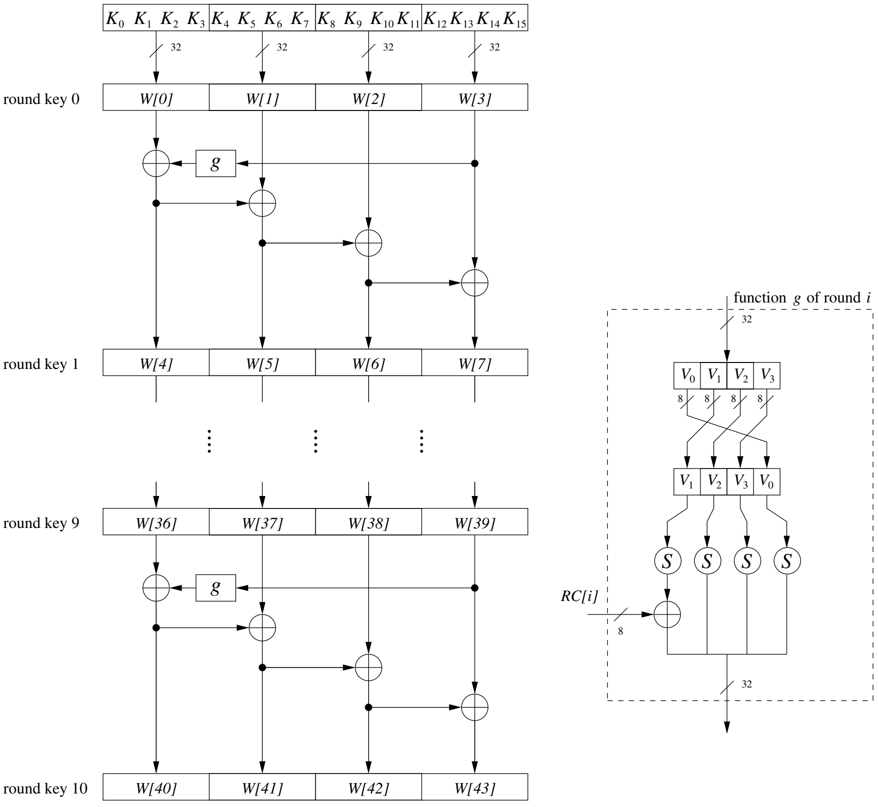

Fig. 4.5 AES key schedule for 128-bit key size

computed as follows. As can be seen in the figure, the leftmost word of a subkey $W[4i]$, where $i = 1, \ldots, 10$, is computed as:

$$W[4i] = W[4(i-1)] + g(W[4i-1]).$$

Here $g()$ is a nonlinear function with a four-byte input and output. The remaining three words of a subkey are computed recursively as:

$$W[4i+j] = W[4i+j-1] + W[4(i-1)+j],$$

where $i = 1, \ldots, 10$ and $j = 1, 2, 3$. The function $g()$ rotates its four input bytes, performs a byte-wise S-Box substitution, and adds a *round coefficient RC* to it. The round coefficient is an element of the Galois field $GF(2^8)$, i.e, an 8-bit value. It is only added to the leftmost byte in the function $g()$. The round coefficients vary from round to round according to the following rule:

$$\begin{aligned}
RC[1] &= x^0 = (00000001)_2,\\
RC[2] &= x^1 = (00000010)_2,\\
RC[3] &= x^2 = (00000100)_2,\\
&\ \vdots\\
RC[10] &= x^9 = (00110110)_2.
\end{aligned}$$

The function $g()$ has two purposes. First, it adds nonlinearity to the key schedule. Second, it removes symmetry in AES. Both properties are necessary to thwart certain block cipher attacks.

Key Schedule for 192-Bit Key AES

AES with 192-bit key has 12 rounds and, thus, 13 subkeys of 128 bit each. The subkeys require 52 words, which are stored in the array elements $W[0], \ldots, W[51]$. The computation of the array elements is quite similar to the 128-bit key case and is shown in Fig. 4.6. There are eight iterations of the key schedule. (Note that these key schedule iterations do *not* correspond to the 12 AES rounds.) Each iteration computes six new words of the subkey array W. The subkey for the first AES round is formed by the array elements $(W[0], W[1], W[2], W[3])$, the second subkey by the elements $(W[4], W[5], W[6], W[7])$, and so on. Eight round coefficients $RC[i]$ are needed within the function $g()$. They are computed as in the 128-bit case and range from $RC[1], \ldots, RC[8]$.

Key Schedule for 256-Bit Key AES

AES with 256-bit key needs 15 subkeys. The subkeys are stored in the 60 words $W[0], \ldots, W[59]$. The computation of the array elements is quite similar to the 128-bit key case and is shown in Fig. 4.7. The key schedule has seven iterations, where each iteration computes eight words for the subkeys. (Again, note that these key schedule iterations do not correspond to the 14 AES rounds.) The subkey for the first AES round is formed by the array elements $(W[0], W[1], W[2], W[3])$, the second subkey by the elements $(W[4], W[5], W[6], W[7])$, and so on. There are seven round coefficients $RC[1], \ldots, RC[7]$ within the function $g()$ needed, that are computed as in the 128-bit case. This key schedule also has a function $h()$ with 4-byte input and output. The function applies the S-Box to all four input bytes.

In general, when implementing any of the key schedules, two different approaches exist:

1. Precomputation All subkeys are expanded first into the array W. The encryption (decryption) of a plaintext (ciphertext) is executed afterwards. This approach is often taken in PC and server implementations of AES, where large pieces of data are encrypted under one key. Please note that this approach requires $(n_r + 1) \cdot 16$ bytes of memory, e.g., $11 \cdot 16 = 176$ bytes if the key size is 128 bits. This is the reason

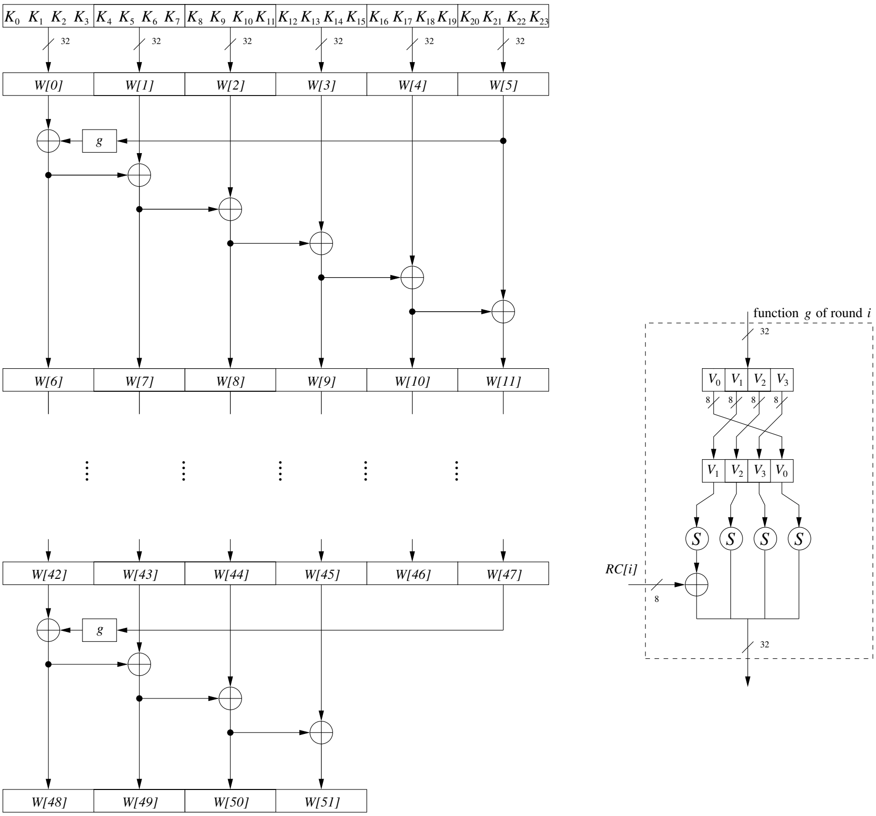

Fig. 4.6 AES key schedule for 192-bit key sizes

why such an implementation on a device with limited memory resources, such as a smart card, is sometimes not desireable.

2. On-the-fly A new subkey is derived for every new round during the encryption (decryption) of a plaintext (ciphertext). Please note that when decrypting ciphertexts, the last subkey is XORed first with the ciphertext. Therefore, it is required to recursively derive all subkeys first and then start with the decryption of a ciphertext and the on-the-fly generation of subkeys. As a result of this overhead, the decryption of a ciphertext is always slightly slower than the encryption of a plaintext when the on-the-fly generation of subkeys is used.

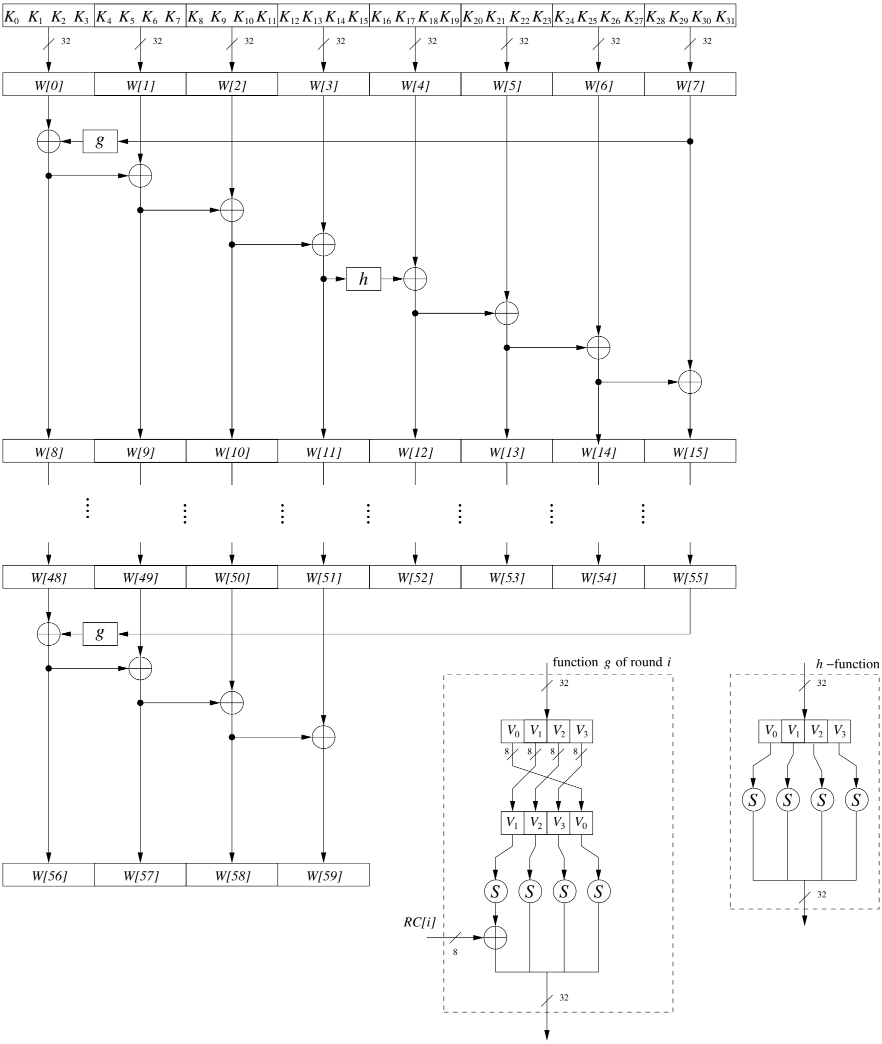
K_0 K_1 K_2 K_3
K_4 K_5 K_6 K_7
K_8 K_9 K_{10} K_{11}
K_{12} K_{13} K_{14} K_{15}
K_{16} K_{17} K_{18} K_{19}
K_{20} K_{21} K_{22} K_{23}
K_{24} K_{25} K_{26} K_{27}
K_{28} K_{29} K_{30} K_{31}
32
W[0] W[1] W[2] W[3] W[4] W[5] W[6] W[7]
g
h
W[8] W[9] W[10] W[11] W[12] W[13] W[14] W[15]
W[48] W[49] W[50] W[51] W[52] W[53] W[54] W[55]
g
W[56] W[57] W[58] W[59]
function g of round i
V_0 V_1 V_2 V_3
8
V_1 V_2 V_3 V_0
S
RC[i]
h -function

Fig. 4.7 AES key schedule for 256-bit key size

4.5 Decryption

Because AES is not based on a Feistel network, all layers must actually be inverted, i.e., the Byte Substitution layer becomes the Inv Byte Substitution layer, the ShiftRows layer becomes the Inv ShiftRows layer, and the MixColumn layer becomes Inv MixColumn layer. However, as we will see, it turns out that the inverse layer operations are fairly similar to the layer operations used for encryption. In addition, the order of the subkeys is reversed, i.e., we need a reversed key schedule. A block diagram of the decryption function is shown in Fig. 4.8.

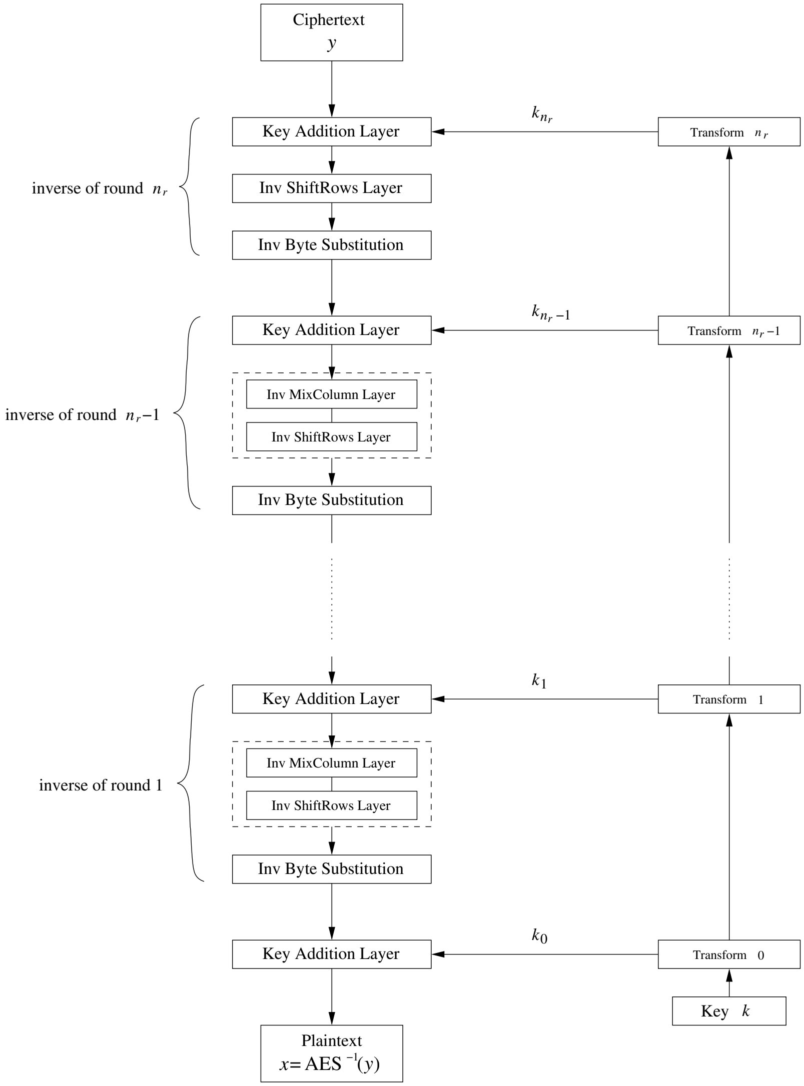

Fig. 4.8 AES decryption block diagram

Since the last encryption round does not perform the MixColum operation, the first decryption round also does not contain the corresponding inverse layer. All other decryption rounds, however, contain all AES layers. In the following, we discuss the inverse layers of the general AES decryption round (Fig. 4.9). Since the

XOR operation is its own inverse, the key addition layer in the decryption mode is the same as in the encryption mode: it consists of a row of plain XOR gates.

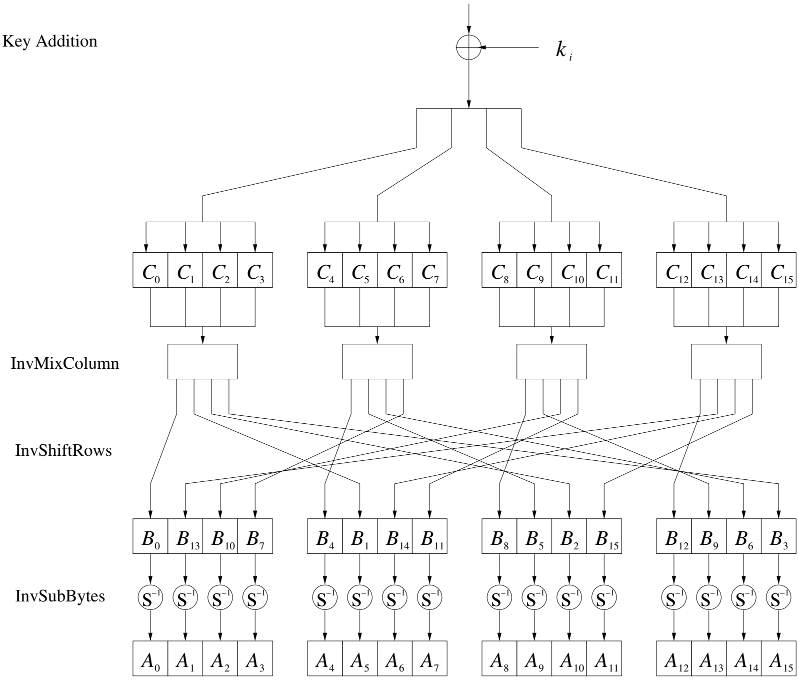
Key Addition
k_i
C_0 C_1 C_2 C_3 C_4 C_5 C_6 C_7 C_8 C_9 C_{10} C_{11} C_{12} C_{13} C_{14} C_{15}
InvMixColumn
InvShiftRows
B_0 B_{13} B_{10} B_7 B_4 B_1 B_{14} B_{11} B_8 B_5 B_2 B_{15} B_{12} B_9 B_6 B_3
InvSubBytes
S^{-1}
A_0 A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10} A_{11} A_{12} A_{13} A_{14} A_{15}

Fig. 4.9 AES decryption round function $1,2,\ldots,n_r-1$

Inverse MixColumn Sublayer

After the addition of the subkey, the inverse MixColumn step is applied to the state (again, the exception is the first decryption round). In order to reverse the MixColumn operation, the inverse of its matrix must be used. The input is a 4-byte column of the State C which is multiplied by the inverse 4×4 matrix. The matrix contains constant entries. Multiplication and addition of the coefficients is done in $GF(2^8)$.

$$\begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{pmatrix} = \begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix}$$

The second column of output bytes (B_4, B_5, B_6, B_7) is computed by multiplying the four input bytes (C_4, C_5, C_6, C_7) by the same constant matrix, and so on. Each value

B_i and C_i is an element from $GF(2^8)$. Also, the constants are elements from $GF(2^8)$. The notation for the constants is hexadecimal and is the same as was used for the MixColumn layer, for example:

$$0B = (0B)_{hex} = (0000\,1011)_2 = x^3 + x + 1.$$

Additions in the vector–matrix multiplication are bitwise XORs.

Inverse ShiftRows Sublayer

In order to reverse the ShiftRows operation of the encryption algorithm, we must shift the rows of the state matrix in the opposite direction. The first row is not changed by the inverse ShiftRows transformation. If the input of the ShiftRows sublayer is given as a state matrix $B = (B_0, B_1, \ldots, B_{15})$:

B_0	B_4	B_8	B_{12}
B_1	B_5	B_9	B_{13}
B_2	B_6	B_{10}	B_{14}
B_3	B_7	B_{11}	B_{15}

the inverse ShiftRows sublayer yields the output:

B_0	B_4	B_8	B_{12}	no shift
B_{13}	B_1	B_5	B_9	⟵ three positions left shift
B_{10}	B_{14}	B_2	B_6	⟵ two positions left shift
B_7	B_{11}	B_{15}	B_3	⟵ one position left shift

Inverse Byte Substitution Layer

The inverse S-Box is used when decrypting a ciphertext. Since the AES S-Box is a bijective, i.e., a one-to-one mapping, it is possible to construct an inverse S-Box such that:

$$A_i = S^{-1}(B_i) = S^{-1}(S(A_i)),$$

where A_i and B_i are elements of the state matrix. The entries of the inverse S-Box are given in Table 4.4.

For readers who are interested in the details of how the entries of inverse S-Box are constructed, we provide a derivation. However, for a functional understanding of AES, the remainder of this section can be skipped. In order to reverse the S-Box substitution, we first have to compute the inverse of the affine transformation. For this, each input byte B_i is considered an element of $GF(2^8)$. The inverse affine transformation on each byte B_i is defined by:

Table 4.4 Inverse AES S-Box: Substitution values in hexadecimal notation for input byte (xy)

		y															
		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
x	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

$$\begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{pmatrix} \equiv \begin{pmatrix} 0&1&0&1&0&0&1&0 \\ 0&0&1&0&1&0&0&1 \\ 1&0&0&1&0&1&0&0 \\ 0&1&0&0&1&0&1&0 \\ 0&0&1&0&0&1&0&1 \\ 1&0&0&1&0&0&1&0 \\ 0&1&0&0&1&0&0&1 \\ 1&0&1&0&0&1&0&0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} \bmod 2,$$

where $(b_7,\ldots,b_0)$ is the bitwise vector representation of $B_i(x)$, and $(b'_7,\ldots,b'_0)$ the result after the inverse affine transformation.

In the second step of the inverse S-Box operation, the Galois field inverse has to be reversed. For this, note that $A_i = (A_i^{-1})^{-1}$. This means that the inverse operation is reversed by computing the inverse again. In our notation we thus have to compute

$$A_i = (B'_i)^{-1} \in GF(2^8)$$

with the fixed reduction polynomial $P(x) = x^8 + x^4 + x^3 + x + 1$. Again, the zero element is mapped to itself. The vector $A_i = (a_7,\ldots,a_0)$ (representing the field element $a_7x^7 + \cdots + a_1x + a_0$) is the result of the substitution:

$$A_i = S^{-1}(B_i).$$

Decryption Key Schedule

Since decryption round one needs the last subkey, the second decryption round needs the second-to-last subkey and so on, we need the subkey in reversed order as shown in Fig. 4.8. In practice this is mainly achieved by computing the entire key schedule first and storing all 11, 13 or 15 subkeys, depending on the number or rounds AES is using (which in turn depends on the three key lengths supported by AES). This precomputation adds usually a small latency to the decryption operation relative to encryption.

4.6 Implementation in Software and Hardware

We briefly comment on the efficiency of the AES cipher with respect to software and hardware implementation.

Software

Unlike DES, AES was designed such that an efficient software implementation is possible. A straightforward implementation of AES which directly follows the data path description, such as the description given in this chapter, is well suited for 8-bit processors such as those found on smart cards, but is not particularly efficient on 32-bit or 64-bit machines, which are common in today's PCs. In a naïve implementation, all time-critical functions (Byte Substitution, ShiftRows, MixColumn) operate on individual bytes. Processing 1 byte per instruction is inefficient on modern 32-bit or 64-bit processors.

However, the Rijndael designers proposed a method which results in fast software implementations. The core idea is to merge all round functions (except the rather trivial key addition) into one table look-up. This results in four tables, each of which consists of 256 entries, where each entry is 32 bits wide. These tables are named a *T-Box*. Four table accesses yield 32 output bits of one round. Hence, one round can be computed with 16 table look-ups. On a 1.2-GHz Intel processor, a throughput of 400 Mbit/s (or 50 MByte/s) is possible. The fastest known implementation on a 64-bit Athlon CPU achieves a theoretical throughput of more than 1.6 Gbit/s. However, conventional hard disc encryption tools with AES or an open-source implementation of AES reach a perfomance of a few hundred Mbit/s on similar platforms.

Hardware

Compared to DES, AES requires more hardware resources for an implementation. However, due to the high integration density of modern integrated circuits, AES can be implemented with very high throughputs in modern ASIC or FPGA (field programmable gate array — these are programmable hardware devices) technology. Commercial AES ASICs can exceed throughputs of 10Gbit/sec. Through parallelization of AES encryption units on one chip, the speed can be further increased. It can be said that symmetric encryption with today's ciphers is extremely fast, not only compared to asymmetric cryptosystems but also compared to other algorithms needed in modern communication systems, such as data compression or signal processing schemes.

4.7 Discussion and Further Reading

AES Algorithm and Security A detailed description of the design principles of AES can be found in [52]. This book by the Rijndael inventors describes the design of the block cipher. Recent research in context to AES can be found online in the *AES Lounge* [68]. This website is a dissemination effort within ECRYPT, the Network of Excellence in Cryptology, and is a rich resource of activities around AES. It gives many links to further information and papers regarding implementation and theoretical aspects of AES.

There is currently no analytical attack against AES known which has a complexity less than a brute-force attack. An elegant algebraic description was found [122], which in turn triggered speculations that this could lead to attacks. Subsequent research showed that an attack is, in fact, not feasible. By now, the common assumption is that the approach will not threaten AES. A good summary on algebraic attacks can be found in [43]. In addition, there have been proposals for many other attacks, including square attack, impossible differential attack or related key attack. Again, a good source for further references is the *AES Lounge*.

The standard reference for the mathematics of finite fields is [110]. A very accessible but brief introduction is also given in [19]. The International Workshop on the Arithmetic of Finite Fields (WAIFI), a relatively new workshop series, is concerned with both the applications and the theory of Galois fields [171].

Implementation As mentioned in Sect. 4.6, in most software implementations on modern CPUs special lookup tables are being used (T-Boxes). An early detailed description of the construction of T-Boxes can be found in [51, Sect. 5]. A description of a high-speed software implementation on modern 32-bit and 64-bit CPUs is given in [116, 115]. The bit slicing technique which was developed in the context of DES is also applicable to AES and can lead to very fast code as shown in [117].

A strong indication for the importance of AES was the recent introduction of special AES instructions by Intel in CPUs starting in 2008. The instructions allow these machines to compute the round operation particularly quickly.

There is wealth of literature dealing with hardware implementation of AES. A good introduction to the area of AES hardware architectures is given in [104, Chap. 10]. As an example of the variety of AES implementations, reference [86] describes a very small FPGA implementation with 2.2Mbit/s and a very fast pipelined FPGA implementation with 25Gbit/s. It is also possible to use the DSP blocks (i.e., fast arithmetic units) available on modern FPGAs for AES, which can also yield throughputs beyond 50Mbit/s [63]. The basic idea in all high-speed architectures is to process several plaintext blocks in parallel by means of pipelining. On the other end of the performance spectrum are lightweight architectures which are optimized for applications such as RFID. The basic idea here is to serialize the data path, i.e., one round is processed in several time steps. Good references are [75, 42].

4.8 Lessons Learned

- AES is a modern block cipher which supports three key lengths of 128, 192 and 256 bit. It provides excellent long-term security against brute-force attacks.
- AES has been studied intensively since the late 1990s and no attacks have been found that are better than brute-force.
- AES is not based on Feistel networks. Its basic operations use Galois field arithmetic and provide strong diffusion and confusion.
- AES is part of numerous open standards such as IPsec or TLS, in addition to being the mandatory encryption algorithm for US government applications. It seems likely that the cipher will be the dominant encryption algorithm for many years to come.
- AES is efficient in software and hardware.

Problems

4.1. Since May 26, 2002, the AES (Advanced Encryption Standard) describes the official standard of the US government.

1. The evolutionary history of AES differs from that of DES. Briefly describe the differences of the AES history in comparison to DES.
2. Outline the fundamental events of the developing process.
3. What is the name of the algorithm that is known as AES?
4. Who developed this algorithm?
5. Which block sizes and key lengths are supported by this algorithm?

4.2. For the AES algorithm, some computations are done by Galois Fields (GF). With the following problems, we practice some basic computations.

Compute the multiplication and addition table for the prime field $GF(7)$. A multiplication table is a square (here: 7 × 7) table which has as its rows and columns all field elements. Its entries are the products of the field element at the corresponding row and column. Note that the table is symmetric along the diagonal. The addition table is completely analogous but contains the sums of field elements as entries.

4.3. Generate the multiplication table for the extension field $GF(2^3)$ for the case that the irreducible polynomial is $P(x) = x^3 + x + 1$. The multiplication table is in this case a 8 × 8 table. (Remark: You can do this manually or write a program for it.)

4.4. Addition in $GF(2^4)$: Compute $A(x) + B(x) \bmod P(x)$ in $GF(2^4)$ using the irreducible polynomial $P(x) = x^4 + x + 1$. What is the influence of the choice of the reduction polynomial on the computation?

1. $A(x) = x^2 + 1, B(x) = x^3 + x^2 + 1$
2. $A(x) = x^2 + 1, B(x) = x + 1$

4.5. Multiplication in $GF(2^4)$: Compute $A(x) \cdot B(x) \bmod P(x)$ in $GF(2^4)$ using the irreducible polynomial $P(x) = x^4 + x + 1$. What is the influence of the choice of the reduction polynomial on the computation?

1. $A(x) = x^2 + 1, B(x) = x^3 + x^2 + 1$
2. $A(x) = x^2 + 1, B(x) = x + 1$

4.6. Compute in $GF(2^8)$:

$$(x^4 + x + 1)/(x^7 + x^6 + x^3 + x^2),$$

where the irreducible polynomial is the one used by AES, $P(x) = x^8 + x^4 + x^3 + x + 1$. Note that Table 4.2 contains a list of all multiplicative inverses for this field.

4.7. We consider the field $GF(2^4)$, with $P(x) = x^4 + x + 1$ being the irreducible polynomial. Find the inverses of $A(x) = x$ and $B(x) = x^2 + x$. You can find the inverses either by trial and error, i.e., brute-force search, or by applying the Euclidean algorithm for polynomials. (However, the Euclidean algorithm is only sketched in this chapter.) Verify your answer by multiplying the inverses you determined by A and B, respectively.

4.8. Find all irreducible polynomials

1. of degree 3 over $GF(2)$,
2. of degree 4 over $GF(2)$.

The best approach for doing this is to consider all polynomials of lower degree and check whether they are factors. Please note that we only consider monic irreducible polynomials, i.e., polynomials with the highest coefficient equal to one.

4.9. We consider AES with 128-bit block length and 128-bit key length. What is the output of the first round of AES if the plaintext consists of 128 ones, and the first subkey (i.e., the first subkey) also consists of 128 ones? You can write your final results in a rectangular array format if you wish.

4.10. In the following, we check the *diffusion properties* of AES after a single round. Let $W = (w_0, w_1, w_2, w_3) = (\texttt{0x01000000}, \texttt{0x00000000}, \texttt{0x00000000}, \texttt{0x00000000})$ be the input in 32-bit chunks to a 128-bit AES. The subkeys for the computation of the result of the first round of AES are $W_0, \ldots, W_7$ with 32 bits each are given by

$$\begin{aligned}
W_0 &= (\texttt{0x2B7E1516}),\\
W_1 &= (\texttt{0x28AED2A6}),\\
W_2 &= (\texttt{0xABF71588}),\\
W_3 &= (\texttt{0x09CF4F3C}),\\
W_4 &= (\texttt{0xA0FAFE17}),\\
W_5 &= (\texttt{0x88542CB1}),\\
W_6 &= (\texttt{0x23A33939}),\\
W_7 &= (\texttt{0x2A6C7605}).
\end{aligned}$$

Use this book to figure out how the input is processed in the first round (e.g., S-Boxes). For the solution, you might also want to write a short computer program or use an existing one. In any case, indicate all intermediate steps for the computation of *ShiftRows*, *SubBytes* and *MixColumns*!

1. Compute the output of the first round of AES to the input W and the subkeys $W_0, \ldots, W_7$.
2. Compute the output of the first round of AES for the case that *all* input bits are zero.
3. How many output bits have changed? Remark that we only consider a single round — after every further round, more output bits will be affected (*avalanche effect*).

4.11. The MixColumn transformation of AES consists of a matrix–vector multiplication in the field $GF(2^8)$ with $P(x) = x^8 + x^4 + x^3 + x + 1$. Let $b = (b_7x^7 + \ldots + b_0)$ be one of the (four) input bytes to the vector–matrix multiplication. Each input byte is multiplied with the constants 01, 02 and 03. Your task is to provide exact equations for computing those three constant multiplications. We denote the result by $d = (d_7x^7 + \ldots + d_0)$.

1. Equations for computing the 8 bits of $d = 01 \cdot b$.
2. Equations for computing the 8 bits of $d = 02 \cdot b$.
3. Equations for computing the 8 bits of $d = 03 \cdot b$.

Note: The AES specification uses “01” to represent the polynomial 1, “02” to represent the polynomial x, and “03” to represent $x + 1$.

4.12. We now look at the gate (or bit) complexity of the MixColumn function, using the results from problem 4.11. We recall from the discussion of stream ciphers that a 2-input XOR gate performs a $GF(2)$ addition.

1. How many 2-input XOR gates are required to perform one constant multiplication by 01, 02 and 03, respectively, in $GF(2^8)$.
2. What is the overall gate complexity of a hardware implementation of one matrix–vector multiplication?
3. What is the overall gate complexity of a hardware implementation of the entire Diffusion layer? We assume permutations require no gates.

4.13. We consider the first part of the ByteSub operation, i.e, the Galois field inversion.

1. Using Table 4.2, what is the inverse of the bytes 29, F3 and 01, where each byte is given in hexadecimal notation?
2. Verify your answer by performing a $GF(2^8)$ multiplication with your answer and the input byte. Note that you have to represent each byte first as polynomials in $GF(2^8)$. The MSB of each byte represents the x^7 coefficient.

4.14. Your task is to compute the S-Box, i.e., the ByteSub, values for the input bytes 29, F3 and 01, where each byte is given in hexadecimal notation.

1. First, look up the inverses using Table 4.2 to obtain values B'. Now, perform the affine mapping by computing the matrix–vector multiplication and addition.
2. Verify your result using the S-Box Table 4.3.
3. What is the value of $S(0)$?

4.15. *Derive* the bit representation for the following round constants within the key schedule:

- $RC[8]$
- $RC[9]$
- $RC[10]$

4.16. For the following, we assume AES with 192-bit key length. Furthermore, let us assume an ASIC which can check $3 \cdot 10^7$ keys per second.

1. If we use 100,000 such ICs in parallel, how long does an average key search take? Compare this period of time with the age of the universe (approx. 10^{10} years).
2. Assume Moore's Law will still be valid for the next few years, how many years do we have to wait until we can build a key search machine to perform an average key search of AES-192 in 24 hours? Again, assume that we use 100,000 ICs in parallel.