Check for updates

FIPS 197

Federal Information Processing Standards Publication

Advanced Encryption Standard (AES)

Category: Computer Security **Subcategory: Cryptography**

Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-8900

This publication is available free of charge from:
https://doi.org/10.6028/NIST.FIPS.197-upd1

Published November 26, 2001; Updated May 9, 2023

U.S. Department of Commerce
Donald L. Evans, Secretary

Technology Administration
Phillip J. Bond, Under Secretary for Technology

National Institute of Standards and Technology
Karen H. Brown, Acting Director

Foreword

The Federal Information Processing Standards Publication Series of the National Institute of Standards and Technology is the official series of publications relating to standards and guidelines developed under 15 U.S.C. 278g-3, and issued by the Secretary of Commerce under 40 U.S.C. 11331.

Comments concerning this Federal Information Processing Standard publication are welcomed and should be submitted using the contact information in the "Inquiries and comments" clause of the announcement section.

James A. St. Pierre, Acting Director
Information Technology Laboratory

Abstract

In 2000, NIST announced the selection of the Rijndael block cipher family as the winner of the Advanced Encryption Standard (AES) competition. Block ciphers are the foundation for many cryptographic services, especially those that provide assurance of the confidentiality of data.

Three members of the Rijndael family are specified in this Standard: AES-128, AES-192, and AES-256. Each of them transforms data in blocks of 128 bits, and the numerical suffix indicates the bit length of the associated cryptographic keys.

Keywords: AES; block cipher; confidentiality; cryptography; encryption; Rijndael.

Federal Information Processing Standards Publication 197

Published: November 26, 2001
Updated: May 9, 2023

Announcing the ADVANCED ENCRYPTION STANDARD (AES)

Federal Information Processing Standards Publications (FIPS) are developed by NIST under 15 U.S.C. 278g-3 and issued by the Secretary of Commerce under 40 U.S.C. 11331.

1. **Name of Standard.** Advanced Encryption Standard (AES) (FIPS 197).

2. **Category of Standard.** Computer Security Standard, Cryptography.

3. **Explanation.** The Advanced Encryption Standard (AES) specifies a FIPS-approved cryptographic algorithm that can be used to protect electronic data. The AES algorithm is a symmetric block cipher that can encrypt (encipher) and decrypt (decipher) digital information.

   The AES algorithm is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits.

4. **Approving Authority.** Secretary of Commerce.

5. **Maintenance Agency.** Department of Commerce, National Institute of Standards and Technology, Information Technology Laboratory (ITL).

6. **Applicability.** Federal Information Processing Standards apply to information systems used or operated by federal agencies, a contractor of an agency, or other organization on behalf of an agency. They do not apply to national security systems as defined in 44 U.S.C. 3552.

   This Standard may be used by federal agencies to protect information when they have determined that encryption is appropriate, in accordance with applicable Office of Management and Budget and agency policies. Federal agencies may also use alternative methods that NIST has indicated are appropriate for this purpose.

   This Standard may be adopted and used by non-Federal Government organizations.

7. **Specifications.** Federal Information Processing Standard (FIPS) 197, Advanced Encryption Standard (AES) (affixed).

8. **Implementations.** The algorithm specified in this Standard may be implemented in software, firmware, hardware, or any combination thereof. The specific implementation may depend on several factors, such as the application, the environment, the technology used, etc. The algorithm shall be used in conjunction with a FIPS-approved or NIST-recommended mode of operation. Object Identifiers (OIDs) and any associated parameters for AES used in

these modes are available at the Computer Security Objects Register (CSOR), located at https://csrc.nist.gov/projects/csor.

NIST has developed a validation program to test implementations for conformance to the algorithms in this Standard. Information about the validation program is available at https://nist.gov/cmvp. Examples for each key size are available at https://csrc.nist.gov/projects/aes.

9. **Implementation Schedule.** This Standard became effective on May 26, 2002.

10. **Patents.** Implementations of the algorithm specified in this Standard may be covered by U.S. and foreign patents.

11. **Export Control.** Certain cryptographic devices and technical data regarding them are subject to federal export controls. Exports of cryptographic modules implementing this Standard and technical data regarding them must comply with all federal laws and regulations and must be licensed by the Bureau of Industry and Security of the U.S. Department of Commerce. Information about export regulations is available at https://www.bis.doc.gov.

12. **Qualifications.** NIST will continue to follow developments in the analysis of the AES algorithm. As with its other cryptographic algorithm standards, NIST will formally reevaluate this Standard every five years.

    Both this Standard and possible threats reducing the security provided through the use of this Standard will undergo review by NIST as appropriate, taking into account newly available analysis and technology. In addition, the awareness of any breakthrough in technology or any mathematical weakness of the algorithm will cause NIST to reevaluate this Standard and provide necessary revisions.

13. **Where to Obtain Copies.** This publication is available by accessing https://csrc.nist.gov/publications. Other computer security publications are available at the same website.

14. **Inquiries and Comments.** Inquiries and comments about this FIPS may be submitted to fips-197-comments@nist.gov.

15. **How to Cite This Publication.** NIST has assigned **NIST FIPS 197-upd1** as the publication identifier for this FIPS, per the NIST Technical Series Publication Identifier Syntax. NIST recommends that it be cited as follows:

    > National Institute of Standards and Technology (2001) Advanced Encryption Standard (AES). (Department of Commerce, Washington, D.C.), Federal Information Processing Standards Publication (FIPS) NIST FIPS 197-upd1, updated May 9, 2023. https://doi.org/10.6028/NIST.FIPS.197-upd1

Federal Information Processing Standards Publication 197

Specification for the ADVANCED ENCRYPTION STANDARD (AES)

Table of Contents

List of Tables

List of Figures

List of Algorithms

1. Introduction

A *block* is a sequence of bits of a given fixed length. A *block cipher* is a family of permutations of blocks that is parameterized by a sequence of bits called the *key*.

In 1997, NIST initiated the Advanced Encryption Standard (AES) development effort [1] and called for the public to submit candidate algorithms for block ciphers. Block ciphers are the foundation for many cryptographic services, especially those that provide assurance of the confidentiality of data. In 2000, NIST announced the selection of Rijndael [2, 3] for the AES.

This Standard specifies three instantiations of Rijndael: AES-128, AES-192, and AES-256, where the suffix indicates the bit length of the key. The block size (i.e., the length of the data inputs and outputs) is 128 bits in each case. Rijndael supports additional block sizes and key lengths that are not adopted in this Standard.

This Standard is organized as follows:

- Section 2 defines the terms, acronyms, algorithm parameters, symbols, and functions in this Standard.
- Section 3 describes the notation and conventions for the ordering and indexing of bits, bytes, and words.
- Section 4 explains some mathematical components of the AES specifications: finite field arithmetic and multiplication by a fixed matrix of finite field elements.
- Section 5 specifies AES-128, AES-192, and AES-256.
- Section 6 provides implementation guidelines on key length requirements, keying restrictions, parameter extensions, and implementation suggestions regarding various platforms.
- Appendix A gives examples of the key expansion routines for AES-128, AES-192, and AES-256.
- Appendix B gives a step-by-step example of an invocation of AES-128.
- Appendix C gives a reference to the NIST website for extensive example vectors for AES-128, AES-192, and AES-256.
- Appendix D summarizes the updates to the original version of this publication.

2. Definitions

2.1 Terms and Acronyms

The following definitions are used in this Standard:

AES	Advanced Encryption Standard.
Affine transformation	A transformation consisting of multiplication by a matrix, followed by the addition of a vector.
Array	A fixed-size data structure that stores a collection of elements, where each element is identified by its integer index or indices.
Bit	A binary digit: 0 or 1.
Block	A sequence of bits of a given fixed length. In this Standard, blocks consist of 128 bits, sometimes represented as arrays of bytes or words.
Block cipher	A family of permutations of blocks that is parameterized by the key.
Byte	A sequence of eight bits.
Equivalent inverse cipher	An alternative specification of the inverse of CIPHER() with a structure similar to that of CIPHER() and with a modified key schedule as input.
Key	The parameter of a block cipher that determines the selection of a permutation from the block cipher family.
Key schedule	The sequence of round keys that are generated from the key by KEYEXPANSION().
Rijndael	The block cipher that NIST selected as the winner of the AES competition.
Round	A sequence of transformations of the state that is iterated *Nr* times in the specifications of CIPHER(), INVCIPHER(), and EQINVCIPHER(). The sequence consists of four transformations, except for one iteration, in which one of the transformations is omitted.
Round key	One of the $Nr+1$ arrays of four words that are derived from the block cipher key using the key expansion routine; each round key is an input to an instance of ADDROUNDKEY() in the AES block cipher.
State	Intermediate result of the AES block cipher that is represented as a two-dimensional array of bytes with four rows and *Nb* columns.
S-box	A non-linear substitution table used in SUBBYTES() and KEYEXPANSION() to perform a one-to-one substitution of a byte value.
Word	A group of 32 bits that is treated either as a single entity or as an array of 4 bytes.

2.2 List of Functions

The following functions are specified in this Standard:

ADDROUNDKEY()	The transformation of the state in which a round key is combined with the state.
AES-128()	The block cipher specified in this Standard with 128-bit keys.
AES-192()	The block cipher specified in this Standard with 192-bit keys.
AES-256()	The block cipher specified in this Standard with 256-bit keys.
CIPHER()	The transformation of blocks that underlies AES-128, AES-192, and AES-256; the key schedule and the number of rounds are parameters of the transformation.
EQINVCIPHER()	The inverse of CIPHER() in which *dw* replaces *w* as the key schedule parameter.
INVCIPHER()	The inverse of CIPHER().
INVMIXCOLUMNS()	The inverse of MIXCOLUMNS().
INVSBOX()	The inverse of SBOX().
INVSHIFTROWS()	The inverse of SHIFTROWS().
INVSUBBYTES()	The inverse of SUBBYTES().
KEYEXPANSION()	The routine that generates the round keys from the key.
KEYEXPANSIONEIC()	The routine that generates the modified round keys for the equivalent inverse cipher.
MIXCOLUMNS()	The transformation of the state that takes all of the columns of the state and mixes their data (independently of one another) to produce new columns.
ROTWORD()	The transformation of words in which the four bytes of the word are permuted cyclically.
SBOX()	The transformation of bytes defined by the S-box.
SHIFTROWS()	The transformation of the state in which the last three rows are cyclically shifted by different offsets.
SUBBYTES()	The transformation of the state that applies the S-box independently to each byte of the state.
SUBWORD()	The transformation of words in which the S-box is applied to each of the four bytes of the word.
XTIMES()	The transformation of bytes in which the polynomial representation of the input byte is multiplied by x, modulo $m(x)$, to produce the polynomial representation of the output byte.

2.3 Algorithm Parameters and Symbols

b^{-1}	The multiplicative inverse of the element b in GF(2^8).
$\tilde{b}$	The input to the affine transformation in the AES S-box.
dw	Word array for the key schedule that is input to the equivalent inverse cipher.
GF(2)	Finite field with two elements.
GF(2^8)	Finite field with 256 elements.
in	The data input to CIPHER() or INVCIPHER(), represented as an array of 16 bytes indexed from 0 to 15.
$m(x)$	The modulus specified in this standard for the polynomial representation of bytes as elements of GF(2^8).
key	The array of Nk words that comprise the key for AES-128, AES-192, or AES-256.
Nb	The number of columns comprising the state, where each column is a 32-bit word. For this Standard, $Nb = 4$.
Nk	The number of 32-bit words comprising the key. Nk is assigned to 4, 6, and 8 for AES-128, AES-192, and AES-256, respectively. (see Section 6.3).
Nr	The number of rounds. Nr is assigned to 10, 12, and 14 for AES-128, AES-192, and AES-256, respectively.
out	The data output of CIPHER() or INVCIPHER(), represented as an array of 16 bytes indexed from 0 to 15.
$Rcon$	Word array for the round constant.
$state$	The state, represented as a two-dimensional array of 16 bytes, with rows and columns indexed from 0 to 3.
$u[i]$	For a one-dimensional array u of words or bytes, the element in the array that is indexed by a non-negative integer i.
$u[i..i+3]$	For an array u of words, the sequence $u[i], u[i+1], u[i+2], u[i+3]$.
w	Word array for the key schedule.
$\oplus$	Either the exclusive-OR operation on bits, the bitwise exclusive-OR operation on bytes, or the bitwise exclusive-OR operation on words.
$\bullet$	Multiplication in $\mathrm{GF}\left(2^8\right)$.
$*$	Integer multiplication.
$\leftarrow$	Assignment of a variable in pseudocode.
$\{\}$	Delimiters for a byte in hexadecimal or binary notation.

3. Notation and Conventions

3.1 Inputs and Outputs

A *bit* is a binary digit — 0 or 1. A *block* is a sequence of 128 bits; the data input and output for the AES block ciphers are blocks. Another input to the AES block ciphers, called the *key*, is a bit sequence that is typically established beforehand and maintained across many invocations of the block cipher. The lengths of the keys for AES-128, AES-192, and AES-256 are 128 bits, 192 bits, and 256 bits, respectively.

3.2 Bytes

The basic processing unit in the AES algorithms is the *byte* — a sequence of eight bits.

A byte value is denoted by the concatenation of the eight bits between braces $\big($e.g., $\{\texttt{10100011}\}\big)$. When the bits of a byte are denoted by an indexed variable, the convention in this Standard is for the indices to decrease from left to right $\big($i.e., $\{b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0\}\big)$.

It is also convenient to denote byte values using hexadecimal notation. The 16 hexadecimal characters represent sequences of four bits, as listed in Table 1. A byte is represented by an ordered pair of hexadecimal characters, where the left character in the pair represents the four left-most bits(i.e., b_7, b_6, b_5, b_4), and the right character in the pair represents the four right-most bits (i.e., b_3, b_2, b_1, b_0). For example, the hexadecimal form of the byte $\{\texttt{10100011}\}$ is $\{\texttt{a3}\}$.

Table 1. Hexadecimal representation of 4-bit sequences

Sequence	0000	0001	0010	0011	0100	0101	0110	0111
Character	0	1	2	3	4	5	6	7
Sequence	1000	1001	1010	1011	1100	1101	1110	1111
Character	8	9	a	b	c	d	e	f

3.3 Indexing of Byte Sequences

In order to unambiguously represent the data and key inputs as sequences of bytes, the following indexing convention is adopted in this Standard. Given a sequence of $8k$ bits,

$$r_0\ r_1\ r_2\ \ldots\ r_{(8k-3)}\ r_{(8k-2)}\ r_{(8k-1)} \tag{3.1}$$

(for some positive integer k), the bytes a_j for $0 \le j \le k-1$ are defined as follows:

$$a_j = \{r_{8j}\ r_{(8j+1)}\ \ldots\ r_{(8j+7)}\}. \tag{3.2}$$

Thus, for example, the data block

$$r_0\ r_1\ r_2\ \ldots\ r_{125}\ r_{126}\ r_{127} \tag{3.3}$$

is represented by the byte sequence

$$a_0\, a_1\, a_2\, \ldots\, a_{13}\, a_{14}\, a_{15}, \tag{3.4}$$

where

$$\begin{aligned} a_0 &= \{r_0\, r_1\, \ldots\, r_7\}; \\ a_1 &= \{r_8\, r_9\, \ldots\, r_{15}\}; \\ &\vdots \\ a_{15} &= \{r_{120}\, r_{121}\, \ldots\, r_{127}\}. \end{aligned} \tag{3.5}$$

As described in Section 3.2, the bits within any individual byte are indexed in decreasing order from left to right. This ordering is more natural for the finite field arithmetic on bytes that is described in Section 4. The two types of bit indices for byte sequences are illustrated in Table 2.

Table 2. Indices for bytes and bits

Bit index in sequence	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
Byte index	0								1								...
Bit index in byte	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	...

3.4 The State

Internally, the algorithms for the AES block ciphers are performed on a two-dimensional (four-by-four) array of bytes called the *state*. In the state array, denoted by s, each individual byte has two indices: a row index r in the range $0 \le r < 4$ and a column index c in the range $0 \le c < 4$. An individual byte of the state is denoted by either $s_{r,c}$ or $s[r,c]$.

In the specifications for the AES block cipher algorithms in Section 5, the first step is to copy the input array of bytes $in_0,\ in_1,\ \ldots,\ in_{15}$ to the state array s as follows:

$$s[r,c] = in[r+4c] \qquad \text{for } 0 \le r < 4 \text{ and } 0 \le c < 4. \tag{3.6}$$

A sequence of transformations is then applied to the state array, after which its final value is copied to the output array of bytes $out_0,\ out_1,\ \ldots,\ out_{15}$ as follows:

$$out[r+4c] = s[r,c] \qquad \text{for } 0 \le r < 4 \text{ and } 0 \le c < 4. \tag{3.7}$$

The correspondence between the indices of the input and output with the indices of the state array is illustrated in Fig. 1.

$$
\begin{array}{cccc}
\multicolumn{4}{c}{\textit{input bytes}} \\
\hline
in_0 & in_4 & in_8 & in_{12} \\
in_1 & in_5 & in_9 & in_{13} \\
in_2 & in_6 & in_{10} & in_{14} \\
in_3 & in_7 & in_{11} & in_{15} \\
\hline
\end{array}
\longrightarrow
\begin{array}{cccc}
\multicolumn{4}{c}{\textit{state array}} \\
\hline
s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\
s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\
s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\
s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \\
\hline
\end{array}
\longrightarrow
\begin{array}{cccc}
\multicolumn{4}{c}{\textit{output bytes}} \\
\hline
out_0 & out_4 & out_8 & out_{12} \\
out_1 & out_5 & out_9 & out_{13} \\
out_2 & out_6 & out_{10} & out_{14} \\
out_3 & out_7 & out_{11} & out_{15} \\
\hline
\end{array}
$$

Figure 1. State array input and output

3.5 Arrays of Words

A *word* is a sequence of four bytes; a block consists of four words. The four columns of state array s are interpreted as an array v of four words as follows, in the notation of Fig. 1:

$$
v_0 = \begin{pmatrix} s_{0,0} \\ s_{1,0} \\ s_{2,0} \\ s_{3,0} \end{pmatrix}, \quad v_1 = \begin{pmatrix} s_{0,1} \\ s_{1,1} \\ s_{2,1} \\ s_{3,1} \end{pmatrix}, \quad v_2 = \begin{pmatrix} s_{0,2} \\ s_{1,2} \\ s_{2,2} \\ s_{3,2} \end{pmatrix}, \quad v_3 = \begin{pmatrix} s_{0,3} \\ s_{1,3} \\ s_{2,3} \\ s_{3,3} \end{pmatrix}. \tag{3.8}
$$

Thus, the column index c of s becomes the index for v, and the row index r of s becomes the index for the four bytes in each word.

Given a one-dimensional array u of words, $u[i]$ denotes the word that is indexed by i, and the sequence of four words $u[i], u[i+1], u[i+2], u[i+3]$ is denoted by $u[i..i+3]$.

4. Mathematical Preliminaries

For some transformations of the AES algorithms specified in Sec. 5, each byte in the state array is interpreted as one of the 256 elements of a finite field, also known as a Galois Field, denoted by GF(2^8). [1]

In order to define addition and multiplication in GF(2^8), each byte $\{b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0\}$ is interpreted as a polynomial, denoted by $b(x)$, as follows:

$$b(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0. \tag{4.1}$$

For example, $\{\texttt{01100011}\}$ is represented by the polynomial $x^6 + x^5 + x + 1$.

4.1 Addition in GF(2^8)

In order to add two elements in the finite field GF(2^8), the coefficients of the polynomials that represent the elements are added modulo 2 (i.e., with the exclusive-OR operation, denoted by $\oplus$), so that $1 \oplus 1 = 0$, $1 \oplus 0 = 1$, and $0 \oplus 0 = 0$.

Equivalently, two bytes can be added by applying the exclusive-OR operation to each pair of corresponding bits in the bytes. Thus, the sum of $\{a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0\}$ and $\{b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0\}$ is $\{a_7 \oplus b_7\ \ a_6 \oplus b_6\ \ a_5 \oplus b_5\ \ a_4 \oplus b_4\ \ a_3 \oplus b_3\ \ a_2 \oplus b_2\ \ a_1 \oplus b_1\ \ a_0 \oplus b_0\}$. (In Section 5.1.4, this definition is extended to words.)

For example, the following three representations of addition are equivalent:

$$\begin{aligned} &(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2 && \text{(polynomial)} \\ &\{\texttt{01010111}\} \oplus \{\texttt{10000011}\} = \{\texttt{11010100}\} && \text{(binary)} \\ &\{\texttt{57}\} \oplus \{\texttt{83}\} = \{\texttt{d4}\} && \text{(hexadecimal).} \end{aligned} \tag{4.2}$$

Because the coefficients of the polynomials are reduced modulo 2, the coefficient 1 is equivalent to the coefficient –1, so addition is equivalent to subtraction. For example, $x^4 + x^2$ represents the same finite field element as $x^4 - x^2$, $-x^4 + x^2$, and $-x^4 - x^2$. Similarly, the sum of any element with itself is the zero element.

4.2 Multiplication in GF(2^8)

The symbol $\bullet$ denotes multiplication in $\mathrm{GF}\left(2^8\right)$. Conceptually, this multiplication is defined on two bytes in two steps: 1) the two polynomials that represent the bytes are multiplied as polynomials, and 2) the resulting polynomial is reduced modulo the following fixed polynomial:

$$m(x) = x^8 + x^4 + x^3 + x + 1. \tag{4.3}$$

Within both steps, the individual coefficients of the polynomials are reduced modulo 2.

[1] Information about the properties of finite fields can be found in textbooks, such as Michael Artin's *Algebra* [4].

Thus, if $b(x)$ and $c(x)$ represent bytes b and c, then $b \bullet c$ is represented by the following modular reduction of their product as polynomials:

$$b(x)c(x) \mod m(x). \tag{4.4}$$

The modular reduction by $m(x)$ may be applied to intermediate steps in the calculation of $b(x)c(x)$; consequently, it is useful to consider the special case that $c(x) = x$ (i.e., $c = \{\texttt{02}\}$). In particular, the product $b \bullet \{\texttt{02}\}$ can be expressed as a function of b, denoted by $\textsc{xTimes}(b)$, as follows:

$$\textsc{xTimes}(b) = \begin{cases} \{b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0\ 0\} & \text{if } b_7 = 0 \\ \{b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0\ 0\} \oplus \{0\ 0\ 0\ 1\ 1\ 0\ 1\ 1\} & \text{if } b_7 = 1. \end{cases} \tag{4.5}$$

Multiplication by higher powers of x (such as $\{\texttt{04}\}$, $\{\texttt{08}\}$, and $\{\texttt{10}\}$) can be implemented by the repeated application of $\textsc{xTimes}()$. For example, let $b = \{\texttt{57}\}$:

$$\begin{aligned} \{\texttt{57}\} \bullet \{\texttt{01}\} &= \{\texttt{57}\} \\ \{\texttt{57}\} \bullet \{\texttt{02}\} &= \textsc{xTimes}(\{\texttt{57}\}) = \{\texttt{ae}\} \\ \{\texttt{57}\} \bullet \{\texttt{04}\} &= \textsc{xTimes}(\{\texttt{ae}\}) = \{\texttt{47}\} \\ \{\texttt{57}\} \bullet \{\texttt{08}\} &= \textsc{xTimes}(\{\texttt{47}\}) = \{\texttt{8e}\} \\ \{\texttt{57}\} \bullet \{\texttt{10}\} &= \textsc{xTimes}(\{\texttt{8e}\}) = \{\texttt{07}\} \\ \{\texttt{57}\} \bullet \{\texttt{20}\} &= \textsc{xTimes}(\{\texttt{07}\}) = \{\texttt{0e}\} \\ \{\texttt{57}\} \bullet \{\texttt{40}\} &= \textsc{xTimes}(\{\texttt{0e}\}) = \{\texttt{1c}\} \\ \{\texttt{57}\} \bullet \{\texttt{80}\} &= \textsc{xTimes}(\{\texttt{1c}\}) = \{\texttt{38}\}. \end{aligned} \tag{4.6}$$

These products facilitate the computation of any multiple of $\{\texttt{57}\}$. For example, because $\{\texttt{13}\} = \{\texttt{10}\} \oplus \{\texttt{02}\} \oplus \{\texttt{01}\}$, it follows that

$$\begin{aligned} \{\texttt{57}\} \bullet \{\texttt{13}\} &= \{\texttt{57}\} \bullet (\{\texttt{01}\} \oplus \{\texttt{02}\} \oplus \{\texttt{10}\}) \\ &= \{\texttt{57}\} \oplus \{\texttt{ae}\} \oplus \{\texttt{07}\} \\ &= \{\texttt{fe}\}. \end{aligned} \tag{4.7}$$

4.3 Multiplication of Words by a Fixed Matrix

Two transformations – MIXCOLUMNS() and INVMIXCOLUMNS() – in the algorithms for the AES block ciphers can be expressed in terms of matrix multiplication. In particular, a distinct fixed matrix is specified for each transformation. For both matrices, each of the 16 entries of the matrix is a byte of a single specified word, denoted here by $[a_0, a_1, a_2, a_3]$.

Given an input word $[b_0, b_1, b_2, b_3]$ to the transformation, the output word $[d_0, d_1, d_2, d_3]$ is determined by finite field arithmetic as follows:

$$\begin{aligned} d_0 &= (a_0 \bullet b_0) \oplus (a_3 \bullet b_1) \oplus (a_2 \bullet b_2) \oplus (a_1 \bullet b_3) \\ d_1 &= (a_1 \bullet b_0) \oplus (a_0 \bullet b_1) \oplus (a_3 \bullet b_2) \oplus (a_2 \bullet b_3) \\ d_2 &= (a_2 \bullet b_0) \oplus (a_1 \bullet b_1) \oplus (a_0 \bullet b_2) \oplus (a_3 \bullet b_3) \\ d_3 &= (a_3 \bullet b_0) \oplus (a_2 \bullet b_1) \oplus (a_1 \bullet b_2) \oplus (a_0 \bullet b_3). \end{aligned} \tag{4.8}$$

The matrix form of Eq. (4.8) is

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}. \tag{4.9}$$

4.4 Multiplicative Inverses in GF(2^8)

For a byte $b \neq \{\texttt{00}\}$, its multiplicative inverse is the unique byte, denoted by b^{-1}, such that

$$b \bullet b^{-1} = \{\texttt{01}\}. \tag{4.10}$$

The definition of the SUBBYTES() transformation in the specifications of the AES block cipher involves multiplicative inverses in GF(2^8), which can be calculated as follows:

$$b^{-1} = b^{254}. \tag{4.11}$$

Alternatively, let $b(x)$ be the polynomial that represents b. The extended Euclidean algorithm [5] can be applied to $b(x)$ and $m(x)$ to find polynomials $a(x)$ and $c(x)$ such that

$$b(x)a(x) + m(x)c(x) = 1. \tag{4.12}$$

It follows that $a(x)$ is the polynomial that represents b^{-1}.

5. Algorithm Specifications

The general function for executing AES-128, AES-192, or AES-256 is denoted by CIPHER(); its inverse is denoted by INVCIPHER().[2]

The core of the algorithms for CIPHER() and INVCIPHER() is a sequence of fixed transformations of the state called a *round*. Each round requires an additional input called the *round key*; the round key is a block that is usually represented as a sequence of four words (i.e., 16 bytes).

An expansion routine, denoted by KEYEXPANSION(), takes the block cipher key as input and generates the round keys as output. In particular, the input to KEYEXPANSION() is represented as an array of words, denoted by *key*, and the output is an expanded array of words, denoted by *w*, called the *key schedule*.

The block ciphers AES-128, AES-192, and AES-256 differ in three respects: 1) the length of the key; 2) the number of rounds, which determines the size of the required key schedule; and 3) the specification of the recursion within KEYEXPANSION(). For each algorithm, the number of rounds is denoted by Nr, and the number of words of the key is denoted by Nk. (The number of words in the state is denoted by Nb for Rijndael in general; in this Standard, $Nb = 4$.) The specific values of Nk, Nb, and Nr are given in Table 3. No other configurations of Rijndael conform to this Standard.

For implementation issues relating to the key length, block size, and number of rounds, see Section 6.3.

Table 3. Key-Block-Round Combinations

	Key length		**Block size**		**Number of rounds**
	Nk	(in bits)	Nb	(in bits)	Nr
AES-128	4	128	4	128	10
AES-192	6	192	4	128	12
AES-256	8	256	4	128	14

The three inputs to CIPHER() are: 1) the data input *in*, which is a block represented as a linear array of 16 bytes; 2) the number of rounds Nr for the instance; and 3) the round keys. Thus,

$$\begin{aligned} \text{AES-128}(in, key) &= \text{CIPHER}(in, 10, \text{KEYEXPANSION}(key)) \\ \text{AES-192}(in, key) &= \text{CIPHER}(in, 12, \text{KEYEXPANSION}(key)) \\ \text{AES-256}(in, key) &= \text{CIPHER}(in, 14, \text{KEYEXPANSION}(key)). \end{aligned} \tag{5.1}$$

The inverse permutations are defined by replacing CIPHER() with INVCIPHER() in Eq. 5.1.

[2] Informally, these functions are sometimes called "encryption" and "decryption," but neutral terminology is appropriate because there are other applications of block ciphers besides encryption.

The specifications of CIPHER(), KEYEXPANSION(), and INVCIPHER() are given in Sections 5.1, 5.2, and 5.3, respectively.

5.1 CIPHER()

The rounds in the specification of CIPHER() are composed of the following four byte-oriented transformations on the state:

- SUBBYTES() applies a substitution table (S-box) to each byte.
- SHIFTROWS() shifts rows of the state array by different offsets.
- MIXCOLUMNS() mixes the data within each column of the state array.
- ADDROUNDKEY() combines a round key with the state.

The four transformations are specified in Sections 5.1.1–5.1.4. In those specifications, the transformed bit, byte, or block is denoted by appending the symbol $'$ as a superscript on the original variable $\left(\text{i.e., by } b_i', b', s_{i,j}', \text{ or } s'\right)$.

The round keys for ADDROUNDKEY() are generated by KEYEXPANSION(), which is specified in Section 5.2. In particular, the key schedule is represented as an array w of $4*(Nr+1)$ words.

CIPHER() is specified in the pseudocode in Alg. 1.

Algorithm 1 Pseudocode for CIPHER()

```
 1: procedure CIPHER(in, Nr, w)
 2:     state ← in                                          ▷ See Sec. 3.4
 3:     state ← ADDROUNDKEY(state, w[0..3])                 ▷ See Sec. 5.1.4
 4:     for round from 1 to Nr − 1 do
 5:         state ← SUBBYTES(state)                         ▷ See Sec. 5.1.1
 6:         state ← SHIFTROWS(state)                        ▷ See Sec. 5.1.2
 7:         state ← MIXCOLUMNS(state)                       ▷ See Sec. 5.1.3
 8:         state ← ADDROUNDKEY(state, w[4 ∗ round..4 ∗ round + 3])
 9:     end for
10:     state ← SUBBYTES(state)
11:     state ← SHIFTROWS(state)
12:     state ← ADDROUNDKEY(state, w[4 ∗ Nr..4 ∗ Nr + 3])
13:     return state                                        ▷ See Sec. 3.4
14: end procedure
```

The first step (Line 2) is to copy the input into the state array using the conventions from Sec. 3.4. After an initial round key addition (Line 3), the state array is transformed by Nr applications of the round function (Lines 4–12); the final round (Lines 10–12) differs in that the MIXCOLUMNS() transformation is omitted. The final state is then returned as the output (Line 13), as described in Section 3.4.

5.1.1 SUBBYTES()

SUBBYTES() is an invertible, non-linear transformation of the state in which a substitution table, called an S-box, is applied independently to each byte in the state. The AES S-box is denoted by SBOX().

Let b denote an input byte to SBOX(), and let c denote the constant byte {01100011}. The output byte $b^{'} = \text{SBOX}(b)$ is constructed by composing the following two transformations:

1. Define an intermediate value $\tilde{b}$, as follows, where b^{-1} is the multiplicative inverse of b, as described in Section 4.4:

$$\tilde{b} = \begin{cases} \{00\} & \text{if } b = \{00\} \\ b^{-1} & \text{if } b \neq \{00\}. \end{cases} \tag{5.2}$$

2. Apply the following affine transformation of the bits of $\tilde{b}$ to produce the bits of $b^{'}$:

$$b'_i = \tilde{b}_i \oplus \tilde{b}_{(i+4) \bmod 8} \oplus \tilde{b}_{(i+5) \bmod 8} \oplus \tilde{b}_{(i+6) \bmod 8} \oplus \tilde{b}_{(i+7) \bmod 8} \oplus c_i. \tag{5.3}$$

The matrix form of Eq. (5.3) is given by Eq. (5.4) below:

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \tilde{b}_0 \\ \tilde{b}_1 \\ \tilde{b}_2 \\ \tilde{b}_3 \\ \tilde{b}_4 \\ \tilde{b}_5 \\ \tilde{b}_6 \\ \tilde{b}_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{5.4}$$

Figure 2 illustrates how SUBBYTES() transforms the state.

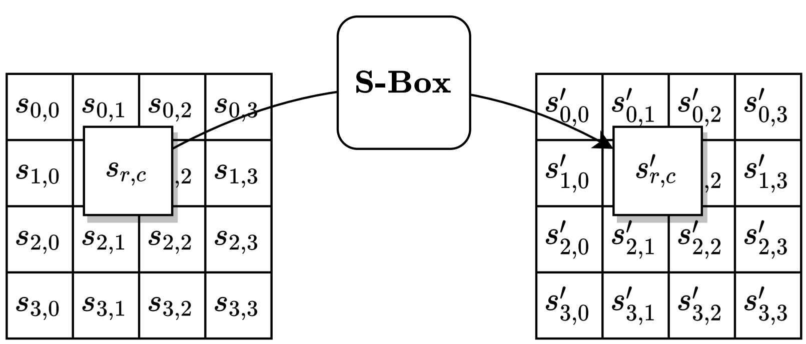

Figure 2. Illustration of SUBBYTES()

The AES S-box is presented in hexadecimal form in Table 4. For example, if $s_{r,c} = \{53\}$, then

Table 4. SBOX(): substitution values for the byte `xy` (in hexadecimal format)

x \ 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
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

the substitution value would be determined by the intersection of the row with index ‘5’ and the column with index ‘3’ in Table 4, so that $s'_{r,c} = \{\texttt{ed}\}$.

5.1.2 SHIFTROWS()

SHIFTROWS() is a transformation of the state in which the bytes in the last three rows of the state are cyclically shifted. The number of positions by which the bytes are shifted depends on the row index r, as follows:

$$s'_{r,c} = s_{r,(c+r) \bmod 4} \qquad \text{for } 0 \leq r < 4 \text{ and } 0 \leq c < 4. \tag{5.5}$$

SHIFTROWS() is illustrated in Figure 3. In that representation of the state, the effect is to move each byte by r positions to the left in the row, cycling the left-most r bytes around to the right end of the row. The first row, where $r = 0$, is unchanged.

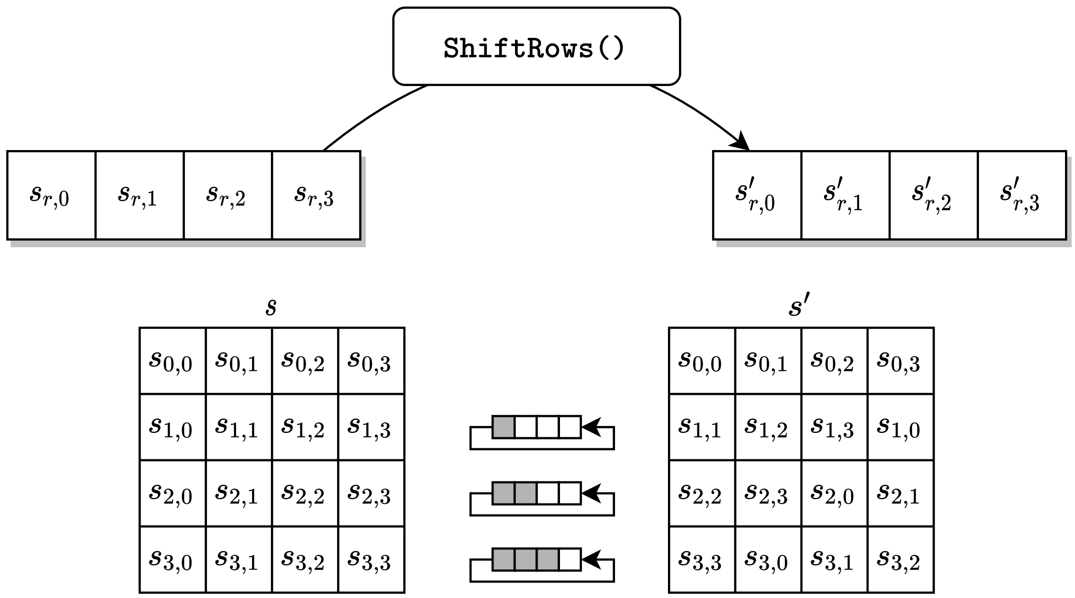

Figure 3. Illustration of SHIFTROWS()

5.1.3 MIXCOLUMNS()

MIXCOLUMNS() is a transformation of the state that multiplies each of the four columns of the state by a single fixed matrix, as described in Section 4.3, with its entries taken from the following word:

$$[a_0, a_1, a_2, a_3] = [\{02\}, \{01\}, \{01\}, \{03\}]. \tag{5.6}$$

Thus,

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < 4, \tag{5.7}$$

so that the individual output bytes are defined as follows:

$$\begin{aligned} s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\ s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c} \\ s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}) \\ s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}). \end{aligned} \tag{5.8}$$

Figure 4 illustrates MIXCOLUMNS().

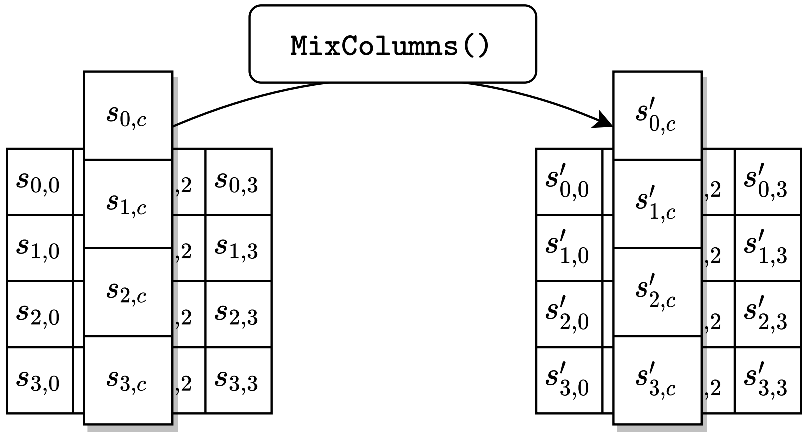

Figure 4. Illustration of MIXCOLUMNS()

5.1.4 ADDROUNDKEY()

ADDROUNDKEY() is a transformation of the state in which a round key is combined with the state by applying the bitwise XOR operation. In particular, each round key consists of four words from the key schedule (described in Section 5.2), each of which is combined with a column of the state as follows:

$$[s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}] = [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}] \oplus [w_{(4*round+c)}] \quad \text{for } 0 \leq c < 4 \tag{5.9}$$

where *round* is a value in the range $0 \leq round \leq Nr$, and $w[i]$ is the array of key schedule words described in Section 5.2. In the specification of CIPHER(), ADDROUNDKEY() is invoked $Nr+1$ times — once prior to the first application of the round function (see Alg. 1) and once within each of the Nr rounds, when $1 \leq round \leq Nr$.

The action of this transformation is illustrated in Fig. 5, where $l = 4 * round$. The byte address within words of the key schedule was described in Sec. 3.5.

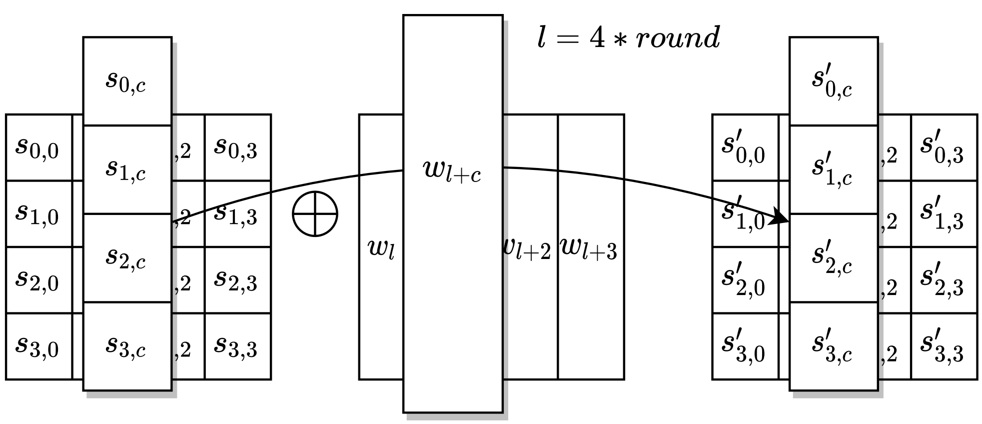

Figure 5. Illustration of ADDROUNDKEY()

5.2 KEYEXPANSION()

KEYEXPANSION() is a routine that is applied to the key to generate $4*(Nr+1)$ words. Thus, four words are generated for each of the $Nr+1$ applications of ADDROUNDKEY() within the specification of CIPHER(), as described in Section 5.1.4. The output of the routine consists of a linear array of words, denoted by $w[i]$, where i is in the range $0 \le i < 4*(Nr+1)$.

KEYEXPANSION() invokes 10 fixed words denoted by $Rcon[j]$ for $1 \le j \le 10$. These 10 words are called the *round constants*. For AES-128, a distinct round constant is called in the generation of each of the 10 round keys. For AES-192 and AES-256, the key expansion routine calls the first eight and seven of these same constants, respectively. The values of $Rcon[j]$ are given in hexadecimal notation in Table 5:

Table 5. Round constants

j	$Rcon[j]$	j	$Rcon[j]$
1	`[01,00,00,00]`	6	`[20,00,00,00]`
2	`[02,00,00,00]`	7	`[40,00,00,00]`
3	`[04,00,00,00]`	8	`[80,00,00,00]`
4	`[08,00,00,00]`	9	`[1b,00,00,00]`
5	`[10,00,00,00]`	10	`[36,00,00,00]`

The value of the left-most byte of $Rcon[j]$ in polynomial form is x^{j-1}. Note that for $j > 0$, these bytes may be generated by successively applying XTIMES() to the byte represented by x^{j-1} (see Eq. 4.5).

Two transformations on words are called within KEYEXPANSION(): ROTWORD() and SUBWORD(). Given an input word represented as a sequence $[a_0, a_1, a_2, a_3]$ of four bytes,

$$\text{ROTWORD}([a_0,a_1,a_2,a_3]) = [a_1,a_2,a_3,a_0], \tag{5.10}$$

and

$$\text{SUBWORD}([a_0,\ldots,a_3]) = [\text{SBOX}(a_0),\text{SBOX}(a_1),\text{SBOX}(a_2),\text{SBOX}(a_3)]. \tag{5.11}$$

The expansion of the key proceeds according to the pseudocode in Alg. 2. The first Nk words of the expanded key are the key itself. Every subsequent word $w[i]$ is generated recursively from the preceding word, $w[i-1]$, and the word Nk positions earlier, $w[i-Nk]$, as follows:

- If i is a multiple of Nk, then $w[i] = w[i-Nk] \oplus \text{SUBWORD}(\text{ROTWORD}(w[i-1])) \oplus Rcon[i/Nk]$.
- For AES-256, if $i+4$ is a multiple of 8, then $w[i] = w[i-Nk] \oplus \text{SUBWORD}(w[i-1])$.
- For all other cases, $w[i] = w[i-Nk] \oplus w[i-1]$.

Algorithm 2 Pseudocode for KEYEXPANSION()

```
procedure KEYEXPANSION(key)
    i ← 0
    while i ≤ Nk − 1 do
        w[i] ← key[4 ∗ i..4 ∗ i + 3]
        i ← i + 1
    end while                                    ▷ When the loop concludes, i = Nk.
    while i ≤ 4 ∗ Nr + 3 do
        temp ← w[i − 1]
        if i mod Nk = 0 then
            temp ← SUBWORD(ROTWORD(temp)) ⊕ Rcon[i/Nk]
        else if Nk > 6 and i mod Nk = 4 then
            temp ← SUBWORD(temp)
        end if
        w[i] ← w[i − Nk] ⊕ temp
        i ← i + 1
    end while
    return w
end procedure
```

Figures 6, 7, and 8 illustrate KEYEXPANSION() for AES-128, AES-192, and AES-256.

5.3 INVCIPHER()

To implement INVCIPHER(), the transformations in the specification of CIPHER() (Section 5.1) are inverted and executed in reverse order. The inverted transformations of the state — denoted by INVSHIFTROWS(), INVSUBBYTES(), INVMIXCOLUMNS(), and ADDROUNDKEY() — are described in Sections 5.3.1–5.3.4.

INVCIPHER() is described in the pseudocode in Alg. 3, where the array w denotes the key schedule, as described in Section 5.2.

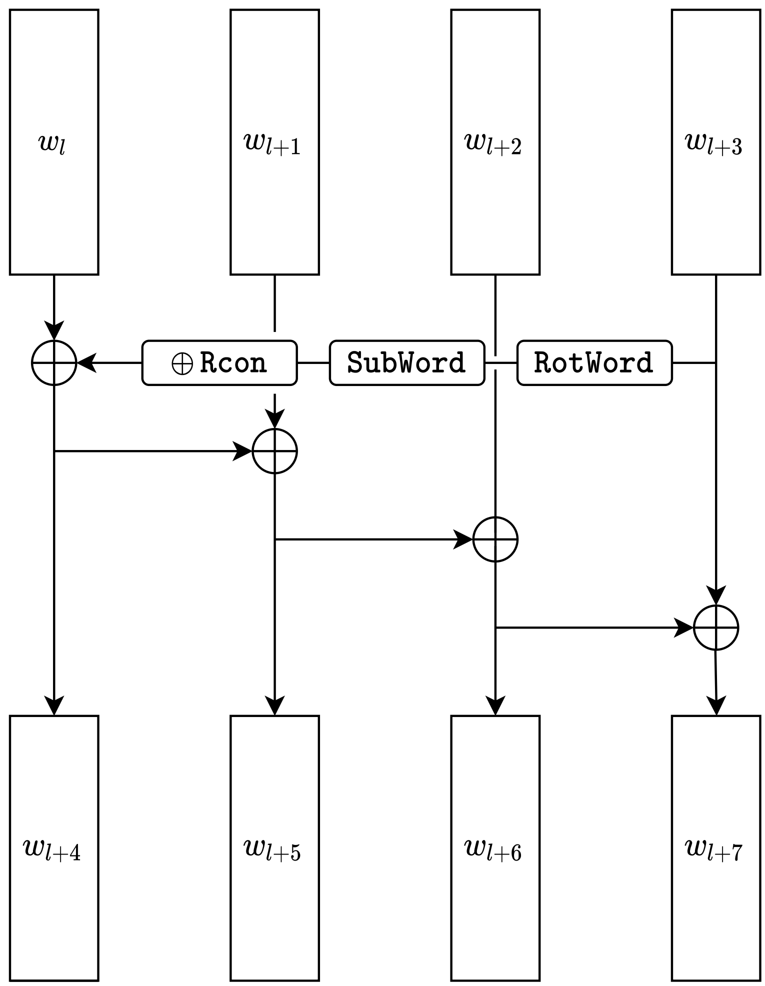

Figure 6. KEYEXPANSION() of AES-128 to generate the words $w[i]$ for $4 \leq i < 44$, where l ranges over the multiples of 4 between 0 and 36

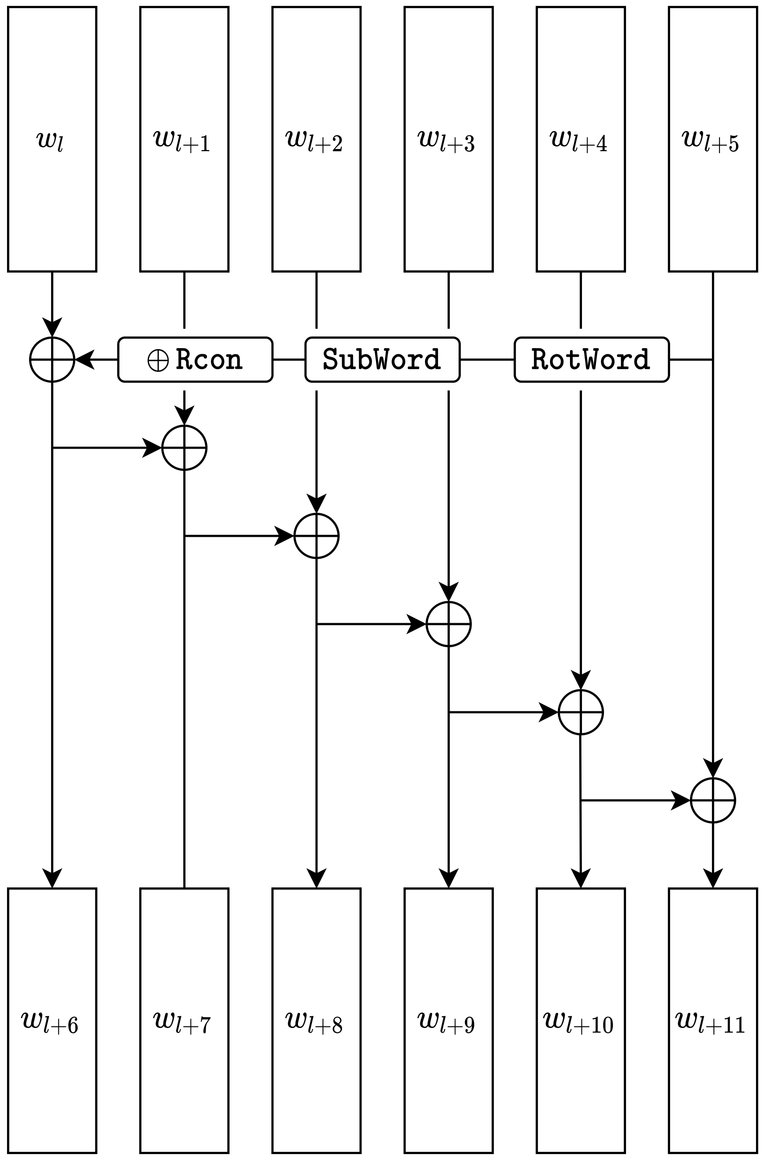

Figure 7. KEYEXPANSION() of AES-192 to generate the words $w[i]$ for $6 \leq i < 52$, where l ranges over the multiples of 6 between 0 and 42

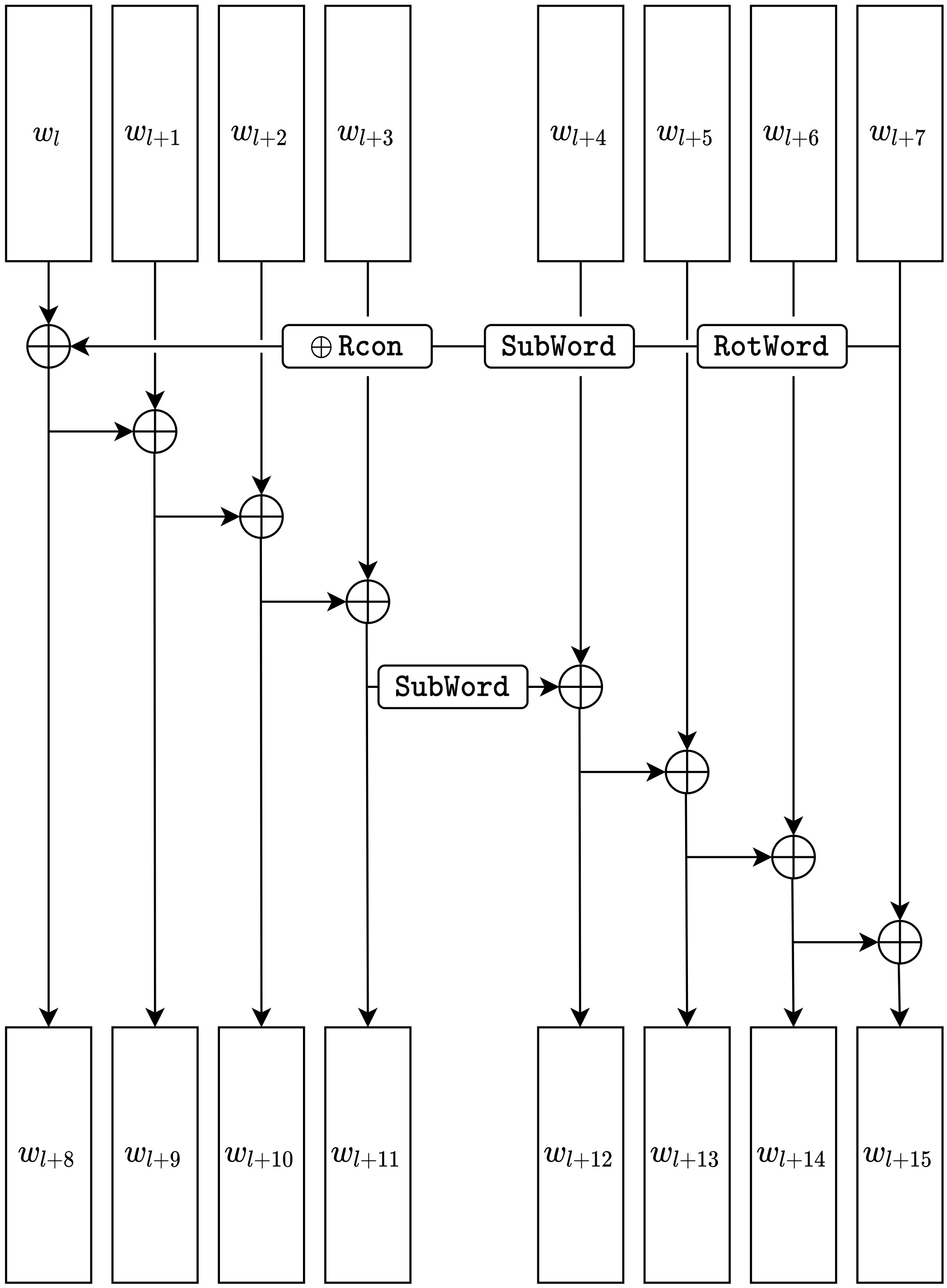

Figure 8. KEYEXPANSION() of AES-256 to generate the words $w[i]$ for $8 \le i < 60$, where l ranges over the multiples of 8 between 0 and 48

Algorithm 3 Pseudocode for INVCIPHER()

```
procedure INVCIPHER(in, Nr, w)
    state ← in                                                  ▷ See Sec. 3.4
    state ← ADDROUNDKEY(state, w[4 ∗ Nr..4 ∗ Nr + 3])           ▷ See Sec. 5.1.4
    for round from Nr − 1 downto 1 do
        state ← INVSHIFTROWS(state)                             ▷ See Sec. 5.3.1
        state ← INVSUBBYTES(state)                              ▷ See Sec. 5.3.2
        state ← ADDROUNDKEY(state, w[4 ∗ round..4 ∗ round + 3])
        state ← INVMIXCOLUMNS(state)                            ▷ See Sec. 5.3.3
    end for
    state ← INVSHIFTROWS(state)
    state ← INVSUBBYTES(state)
    state ← ADDROUNDKEY(state, w[0..3])
    return state
end procedure
```

5.3.1 INVSHIFTROWS()

INVSHIFTROWS() is the inverse of the SHIFTROWS(). In particular, the bytes in the last three rows of the state are cyclically shifted as follows:

$$s'_{r,c} = s_{r,(c-r) \bmod 4} \qquad \text{for } 0 \le r < 4 \text{ and } 0 \le c < 4. \tag{5.12}$$

INVSHIFTROWS() is illustrated in Figure 9. In that representation of the state, the effect is to move each byte by r positions to the right in the row, cycling the right-most r bytes around to the left end of the row. The first row, where $r = 0$, is unchanged.

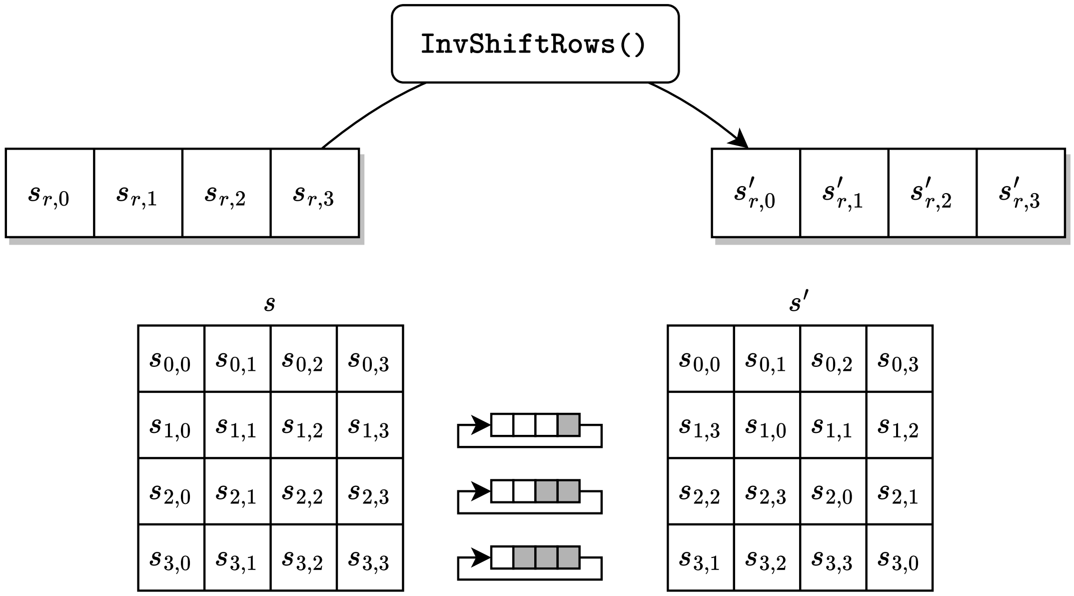

Figure 9. Illustration of INVSHIFTROWS()

5.3.2 INVSUBBYTES()

INVSUBBYTES() is the inverse of SUBBYTES(), in which the inverse of SBOX(), denoted by INVSBOX(), is applied to each byte of the state. INVSBOX() is derived from Table 4 by switching the roles of inputs and outputs, as presented in Table 6:

Table 6. INVSBOX(): substitution values for the byte `xy` (in hexadecimal format)

x \ 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
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

5.3.3 INVMIXCOLUMNS()

INVMIXCOLUMNS() is the inverse of MIXCOLUMNS(). In particular, INVMIXCOLUMNS() multiplies each of the four columns of the state by a single fixed matrix, as described in Section 4.3, with its entries taken from the following word:

$$[a_0, a_1, a_2, a_3] = [\{\texttt{0e}\}, \{\texttt{09}\}, \{\texttt{0d}\}, \{\texttt{0b}\}]. \tag{5.13}$$

Thus,

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} \texttt{0e} & \texttt{0b} & \texttt{0d} & \texttt{09} \\ \texttt{09} & \texttt{0e} & \texttt{0b} & \texttt{0d} \\ \texttt{0d} & \texttt{09} & \texttt{0e} & \texttt{0b} \\ \texttt{0b} & \texttt{0d} & \texttt{09} & \texttt{0e} \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < 4. \tag{5.14}$$

As a result of this matrix multiplication, the four bytes in a column are replaced by the following:

$$\begin{aligned} s'_{0,c} &= (\{\texttt{0e}\} \bullet s_{0,c}) \oplus (\{\texttt{0b}\} \bullet s_{1,c}) \oplus (\{\texttt{0d}\} \bullet s_{2,c}) \oplus (\{\texttt{09}\} \bullet s_{3,c}) \\ s'_{1,c} &= (\{\texttt{09}\} \bullet s_{0,c}) \oplus (\{\texttt{0e}\} \bullet s_{1,c}) \oplus (\{\texttt{0b}\} \bullet s_{2,c}) \oplus (\{\texttt{0d}\} \bullet s_{3,c}) \\ s'_{2,c} &= (\{\texttt{0d}\} \bullet s_{0,c}) \oplus (\{\texttt{09}\} \bullet s_{1,c}) \oplus (\{\texttt{0e}\} \bullet s_{2,c}) \oplus (\{\texttt{0b}\} \bullet s_{3,c}) \\ s'_{3,c} &= (\{\texttt{0b}\} \bullet s_{0,c}) \oplus (\{\texttt{0d}\} \bullet s_{1,c}) \oplus (\{\texttt{09}\} \bullet s_{2,c}) \oplus (\{\texttt{0e}\} \bullet s_{3,c}). \end{aligned} \tag{5.15}$$

5.3.4 Inverse of ADDROUNDKEY()

ADDROUNDKEY(), described in Section 5.1.4, is its own inverse.

5.3.5 EQINVCIPHER()

Several properties of the AES algorithm allow for an alternative specification of the inverse of CIPHER(), called the *equivalent inverse cipher*, denoted by EQINVCIPHER(). In the specification of EQINVCIPHER(), the transformations of the round function of the cipher in Alg. 1 are directly replaced by their inverses in EQINVCIPHER(), in the same order. The efficiency of this structure in comparison to the specification of INVCIPHER() in Alg. 3 is explained in the Rijndael proposal document [2].

The pseudocode for the equivalent inverse cipher, given in Alg. 4, uses a modified key schedule, denoted by the word array *dw*. The routine to generate *dw* is an extension of KEYEXPANSION(), denoted by KEYEXPANSIONEIC(), whose pseudocode is given in Alg. 5.

Algorithm 4 Pseudocode for EQINVCIPHER()

```
procedure EQINVCIPHER(in, Nr, dw)
    state ← in
    state ← ADDROUNDKEY(state, dw[4 ∗ Nr..4 ∗ Nr + 3])
    for round from Nr − 1 downto 1 do
        state ← INVSUBBYTES(state)
        state ← INVSHIFTROWS(state)
        state ← INVMIXCOLUMNS(state)
        state ← ADDROUNDKEY(state, dw[4 ∗ round..4 ∗ round + 3])
    end for
    state ← INVSUBBYTES(state)
    state ← INVSHIFTROWS(state)
    state ← ADDROUNDKEY(state, dw[0..3])
    return state
end procedure
```

Algorithm 5 Pseudocode for KEYEXPANSIONEIC()

```
1:  procedure KEYEXPANSIONEIC(key)
2:      i ← 0
3:      while i ≤ Nk − 1 do
4:          w[i] ← key[4i..4i + 3]
5:          dw[i] ← w[i]
6:          i ← i + 1
7:      end while                          ▷ When the loop concludes, i = Nk.
8:      while i ≤ 4 ∗ Nr + 3 do
9:          temp ← w[i − 1]
10:         if i mod Nk = 0 then
11:             temp ← SUBWORD(ROTWORD(temp)) ⊕ Rcon[i/Nk]
12:         else if Nk > 6 and i mod Nk = 4 then
13:             temp ← SUBWORD(temp)
14:         end if
15:         w[i] ← w[i − Nk] ⊕ temp
16:         dw[i] ← w[i]
17:         i ← i + 1
18:     end while
19:     for round from 1 to Nr − 1 do
20:         i ← 4 ∗ round
21:         dw[i..i + 3] ← INVMIXCOLUMNS(dw[i..i + 3])   ▷ Note change of type.
22:     end for
23:     return dw
24: end procedure
```

The first and last round keys in *dw* are the same as in *w*; the modification of the other round keys is described in Lines 19–22. The comment in Line 21 refers to the input to INVMIXCOLUMNS(): the one-dimensional array of words is converted to a two-dimensional array of bytes, as in Fig. 1.

6. Implementation Considerations

6.1 Key Length Requirements

An implementation of the AES algorithm shall support at least one of the three key lengths specified in Sec. 5: 128, 192, or 256 bits (i.e., $Nk = 4$, 6, or 8, respectively). Implementations may optionally support two or three key lengths, which may promote the interoperability of algorithm implementations.

6.2 Keying Restrictions

When a cryptographic key has been generated appropriately (see NIST Special Publication 800-133, Rev. 2 [6] for guidelines), no restriction is imposed when the resulting key is used for the AES algorithm.

6.3 Parameter Extensions

In Table 3, this Standard explicitly defines the allowed values for the key length (Nk), block size (Nb), and number of rounds (Nr). However, future revisions of this Standard could include changes or additions to the allowed values for those parameters. Therefore, implementers may choose to design their AES implementations with future flexibility in mind.

6.4 Implementation Suggestions Regarding Various Platforms

Implementation variations are possible that may, in many cases, offer performance or other advantages. Given the same input key and data (plaintext or ciphertext), any implementation that produces the same output (ciphertext or plaintext) as the algorithm specified in this Standard is an equivalent implementation of the AES algorithm.

The AES proposal document [2] and other resources located on the AES page [7] include suggestions on how to efficiently implement the AES algorithm on a variety of platforms. Suggested implementations are intended to explain the inner workings of the AES algorithm but do not provide protection against various implementation attacks.

A physical implementation may leak key-dependent information through side channels, such as the time taken to perform a computation, or when faults are injected into the computation. When such attacks are non-invasive, they can be effective even when there are mechanisms to detect physical tampering of the device. For example, cache-timing attacks may affect AES implementations on software platforms that use a cache to accelerate the access to data from main memory.

Protecting implementations of the AES algorithm against implementation attacks where applicable should be considered. Such considerations are outside of the scope of this document but are taken into account when testing for conformance to the algorithm in this Standard according to the validation program developed by NIST (see https://nist.gov/cmvp).

6.5 Modes of Operation

Block cipher modes of operation are cryptographic functions that feature a block cipher to provide information services, such as confidentiality and authentication. NIST-recommended modes of operation are specified in the 800-38 series of NIST Special Publications. Further information is available at https://csrc.nist.gov/Projects/block-cipher-techniques/BCM.

References

[1] James Nechvatal, Elaine Barker, Lawrence Bassham, William Burr, Morris Dworkin, James Foti, and Edward Roback. Report on the Development of the Advanced Encryption Standard (AES). *Journal of Research of NIST (NIST JRES)*, May 2001. https://doi.org/10.6028/jres.106.023.

[2] Joan Daemen and Vincent Rijmen. AES Proposal: Rijndael Document Version 2. AES Algorithm Submission, September 1999. Available at https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf.

[3] Joan Daemen and Vincent Rijmen. *The Design of Rijndael - The Advanced Encryption Standard (AES), Second Edition*. Information Security and Cryptography. Springer, 2020. https://doi.org/10.1007/978-3-662-60769-5.

[4] Michael Artin. *Algebra*. Pearson Modern Classic. Pearson, second edition, 2017.

[5] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., USA, 1st edition, 1997. https://doi.org/10.1201/9780429466335.

[6] Elaine Barker, Allen Roginsky, and Richard Davis. Recommendation for Cryptographic Key Generation. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-133, Rev. 2, June 2020. https://doi.org/10.6028/NIST.SP.800-133r2.

[7] National Institute of Standards and Technology. AES Development, 2022. Available at https://csrc.nist.gov/projects/aes.

[8] National Institute of Standards and Technology. Cryptographic Standards and Guidelines: Examples with Intermediate Values, 2022. Available at https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/example-values.

[9] National Institute of Standards and Technology. Crypto Publications Review Board, 2022. Available at https://csrc.nist.gov/projects/crypto-publication-review-project.

[10] Nicky Mouha. Review of the Advanced Encryption Standard. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Interagency Report (IR) 8319. https://doi.org/10.6028/NIST.IR.8319.

Appendix A — Key Expansion Examples

This appendix shows the development of the key schedule for each key size. Note that multi-byte values are presented using the notation described in Sec. 3. The intermediate values produced during the development of the key schedule (see Sec. 5.2) are given in the following table (all values are in hexadecimal format with the exception of the index column (i)).

A.1 Expansion of a 128-bit Key

This section contains the key expansion of the following key:

```
Key = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c
```

for $Nk = 4$, which results in

w_0 = 2b7e1516 w_1 = 28aed2a6 w_2 = abf71588 w_3 = 09cf4f3c

i (dec)	*temp*	After RotWord()	After SubWord()	$Rcon[i/Nk]$	After XOR with *Rcon*	$w[i-Nk]$	$w[i]$ = *temp* $\oplus$ $w[i-Nk]$
4	09cf4f3c	cf4f3c09	8a84eb01	01000000	8b84eb01	2b7e1516	a0fafe17
5	a0fafe17					28aed2a6	88542cb1
6	88542cb1					abf71588	23a33939
7	23a33939					09cf4f3c	2a6c7605
8	2a6c7605	6c76052a	50386be5	02000000	52386be5	a0fafe17	f2c295f2
9	f2c295f2					88542cb1	7a96b943
10	7a96b943					23a33939	5935807a
11	5935807a					2a6c7605	7359f67f
12	7359f67f	59f67f73	cb42d28f	04000000	cf42d28f	f2c295f2	3d80477d
13	3d80477d					7a96b943	4716fe3e
14	4716fe3e					5935807a	1e237e44
15	1e237e44					7359f67f	6d7a883b
16	6d7a883b	7a883b6d	dac4e23c	08000000	d2c4e23c	3d80477d	ef44a541
17	ef44a541					4716fe3e	a8525b7f
18	a8525b7f					1e237e44	b671253b
19	b671253b					6d7a883b	db0bad00
20	db0bad00	0bad00db	2b9563b9	10000000	3b9563b9	ef44a541	d4d1c6f8
21	d4d1c6f8					a8525b7f	7c839d87
22	7c839d87					b671253b	caf2b8bc
23	caf2b8bc					db0bad00	11f915bc

24	11f915bc	f915bc11	99596582	20000000	b9596582	d4d1c6f8	6d88a37a
25	6d88a37a					7c839d87	110b3efd
26	110b3efd					caf2b8bc	dbf98641
27	dbf98641					11f915bc	ca0093fd
28	ca0093fd	0093fdca	63dc5474	40000000	23dc5474	6d88a37a	4e54f70e
29	4e54f70e					110b3efd	5f5fc9f3
30	5f5fc9f3					dbf98641	84a64fb2
31	84a64fb2					ca0093fd	4ea6dc4f
32	4ea6dc4f	a6dc4f4e	2486842f	80000000	a486842f	4e54f70e	ead27321
33	ead27321					5f5fc9f3	b58dbad2
34	b58dbad2					84a64fb2	312bf560
35	312bf560					4ea6dc4f	7f8d292f
36	7f8d292f	8d292f7f	5da515d2	1b000000	46a515d2	ead27321	ac7766f3
37	ac7766f3					b58dbad2	19fadc21
38	19fadc21					312bf560	28d12941
39	28d12941					7f8d292f	575c006e
40	575c006e	5c006e57	4a639f5b	36000000	7c639f5b	ac7766f3	d014f9a8
41	d014f9a8					19fadc21	c9ee2589
42	c9ee2589					28d12941	e13f0cc8
43	e13f0cc8					575c006e	b6630ca6

A.2 Expansion of a 192-bit Key

This section contains the key expansion of the following key:

```
Key =  8e 73 b0 f7 da 0e 64 52 c8 10 f3 2b
       80 90 79 e5 62 f8 ea d2 52 2c 6b 7b
```

for $Nk = 6$, which results in

$$w_0 = \texttt{8e73b0f7} \quad w_1 = \texttt{da0e6452} \quad w_2 = \texttt{c810f32b}$$
$$w_3 = \texttt{809079e5} \quad w_4 = \texttt{62f8ead2} \quad w_5 = \texttt{522c6b7b}$$

i (dec)	*temp*	After RotWord()	After SubWord()	*Rcon*[*i*/*Nk*]	After XOR with *Rcon*	*w*[*i*−*Nk*]	*w*[*i*] = *temp* ⊕ *w*[*i*−*Nk*]
6	522c6b7b	2c6b7b52	717f2100	01000000	707f2100	8e73b0f7	fe0c91f7
7	fe0c91f7					da0e6452	2402f5a5
8	2402f5a5					c810f32b	ec12068e

9	ec12068e					809079e5	6c827f6b
10	6c827f6b					62f8ead2	0e7a95b9
11	0e7a95b9					522c6b7b	5c56fec2
12	5c56fec2	56fec25c	b1bb254a	02000000	b3bb254a	fe0c91f7	4db7b4bd
13	4db7b4bd					2402f5a5	69b54118
14	69b54118					ec12068e	85a74796
15	85a74796					6c827f6b	e92538fd
16	e92538fd					0e7a95b9	e75fad44
17	e75fad44					5c56fec2	bb095386
18	bb095386	095386bb	01ed44ea	04000000	05ed44ea	4db7b4bd	485af057
19	485af057					69b54118	21efb14f
20	21efb14f					85a74796	a448f6d9
21	a448f6d9					e92538fd	4d6dce24
22	4d6dce24					e75fad44	aa326360
23	aa326360					bb095386	113b30e6
24	113b30e6	3b30e611	e2048e82	08000000	ea048e82	485af057	a25e7ed5
25	a25e7ed5					21efb14f	83b1cf9a
26	83b1cf9a					a448f6d9	27f93943
27	27f93943					4d6dce24	6a94f767
28	6a94f767					aa326360	c0a69407
29	c0a69407					113b30e6	d19da4e1
30	d19da4e1	9da4e1d1	5e49f83e	10000000	4e49f83e	a25e7ed5	ec1786eb
31	ec1786eb					83b1cf9a	6fa64971
32	6fa64971					27f93943	485f7032
33	485f7032					6a94f767	22cb8755
34	22cb8755					c0a69407	e26d1352
35	e26d1352					d19da4e1	33f0b7b3
36	33f0b7b3	f0b7b333	8ca96dc3	20000000	aca96dc3	ec1786eb	40beeb28
37	40beeb28					6fa64971	2f18a259
38	2f18a259					485f7032	6747d26b
39	6747d26b					22cb8755	458c553e
40	458c553e					e26d1352	a7e1466c
41	a7e1466c					33f0b7b3	9411f1df
42	9411f1df	11f1df94	82a19e22	40000000	c2a19e22	40beeb28	821f750a
43	821f750a					2f18a259	ad07d753

44	ad07d753					6747d26b	ca400538
45	ca400538					458c553e	8fcc5006
46	8fcc5006					a7e1466c	282d166a
47	282d166a					9411f1df	bc3ce7b5
48	bc3ce7b5	3ce7b5bc	eb94d565	80000000	6b94d565	821f750a	e98ba06f
49	e98ba06f					ad07d753	448c773c
50	448c773c					ca400538	8ecc7204
51	8ecc7204					8fcc5006	01002202

A.3 Expansion of a 256-bit Key

This section contains the key expansion of the following key:

```
Key =  60 3d eb 10 15 ca 71 be 2b 73 ae f0 85 7d 77 81
       1f 35 2c 07 3b 61 08 d7 2d 98 10 a3 09 14 df f4
```

for $Nk = 8$, which results in

w_0 = 603deb10 w_1 = 15ca71be w_2 = 2b73aef0 w_3 = 857d7781
w_4 = 1f352c07 w_5 = 3b6108d7 w_6 = 2d9810a3 w_7 = 0914dff4

i (dec)	*temp*	After RotWord()	After SubWord()	$Rcon[i/Nk]$	After XOR with *Rcon*	$w[i-Nk]$	$w[i]$ = *temp* $\oplus$ $w[i-Nk]$
8	0914dff4	14dff409	fa9ebf01	01000000	fb9ebf01	603deb10	9ba35411
9	9ba35411					15ca71be	8e6925af
10	8e6925af					2b73aef0	a51a8b5f
11	a51a8b5f					857d7781	2067fcde
12	2067fcde		b785b01d			1f352c07	a8b09c1a
13	a8b09c1a					3b6108d7	93d194cd
14	93d194cd					2d9810a3	be49846e
15	be49846e					0914dff4	b75d5b9a
16	b75d5b9a	5d5b9ab7	4c39b8a9	02000000	4e39b8a9	9ba35411	d59aecb8
17	d59aecb8					8e6925af	5bf3c917
18	5bf3c917					a51a8b5f	fee94248
19	fee94248					2067fcde	de8ebe96
20	de8ebe96		1d19ae90			a8b09c1a	b5a9328a
21	b5a9328a					93d194cd	2678a647
22	2678a647					be49846e	98312229

23	98312229					b75d5b9a	2f6c79b3
24	2f6c79b3	6c79b32f	50b66d15	04000000	54b66d15	d59aecb8	812c81ad
25	812c81ad					5bf3c917	dadf48ba
26	dadf48ba					fee94248	24360af2
27	24360af2					de8ebe96	fab8b464
28	fab8b464		2d6c8d43			b5a9328a	98c5bfc9
29	98c5bfc9					2678a647	bebd198e
30	bebd198e					98312229	268c3ba7
31	268c3ba7					2f6c79b3	09e04214
32	09e04214	e0421409	e12cfa01	08000000	e92cfa01	812c81ad	68007bac
33	68007bac					dadf48ba	b2df3316
34	b2df3316					24360af2	96e939e4
35	96e939e4					fab8b464	6c518d80
36	6c518d80		50d15dcd			98c5bfc9	c814e204
37	c814e204					bebd198e	76a9fb8a
38	76a9fb8a					268c3ba7	5025c02d
39	5025c02d					09e04214	59c58239
40	59c58239	c5823959	a61312cb	10000000	b61312cb	68007bac	de136967
41	de136967					b2df3316	6ccc5a71
42	6ccc5a71					96e939e4	fa256395
43	fa256395					6c518d80	9674ee15
44	9674ee15		90922859			c814e204	5886ca5d
45	5886ca5d					76a9fb8a	2e2f31d7
46	2e2f31d7					5025c02d	7e0af1fa
47	7e0af1fa					59c58239	27cf73c3
48	27cf73c3	cf73c327	8a8f2ecc	20000000	aa8f2ecc	de136967	749c47ab
49	749c47ab					6ccc5a71	18501dda
50	18501dda					fa256395	e2757e4f
51	e2757e4f					9674ee15	7401905a
52	7401905a		927c60be			5886ca5d	cafaaae3
53	cafaaae3					2e2f31d7	e4d59b34
54	e4d59b34					7e0af1fa	9adf6ace
55	9adf6ace					27cf73c3	bd10190d
56	bd10190d	10190dbd	cad4d77a	40000000	8ad4d77a	749c47ab	fe4890d1
57	fe4890d1					18501dda	e6188d0b
58	e6188d0b					e2757e4f	046df344
59	046df344					7401905a	706c631e

Appendix B — Cipher Example

The following diagram shows the values in the state array as the cipher progresses for a block length and a key length of 16 bytes each (i.e., $Nb = 4$ and $Nk = 4$).

```
Input  = 32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34
Key    = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c
```

The Round Key values are taken from the Key Expansion example in Appendix A.1.

Round Number	Start of Round				After SubBytes				After ShiftRows				After MixColumns				Round Key Value			
input	32	88	31	e0													2b	28	ab	09
	43	5a	31	37													7e	ae	f7	cf
	f6	30	98	07													15	d2	15	4f
	a8	8d	a2	34													16	a6	88	3c
1	19	a0	9a	e9	d4	e0	b8	1e	d4	e0	b8	1e	04	e0	48	28	a0	88	23	2a
	3d	f4	c6	f8	27	bf	b4	41	bf	b4	41	27	66	cb	f8	06	fa	54	a3	6c
	e3	e2	8d	48	11	98	5d	52	5d	52	11	98	81	19	d3	26	fe	2c	39	76
	be	2b	2a	08	ae	f1	e5	30	30	ae	f1	e5	e5	9a	7a	4c	17	b1	39	05
2	a4	68	6b	02	49	45	7f	77	49	45	7f	77	58	1b	db	1b	f2	7a	59	73
	9c	9f	5b	6a	de	db	39	02	db	39	02	de	4d	4b	e7	6b	c2	96	35	59
	7f	35	ea	50	d2	96	87	53	87	53	d2	96	ca	5a	ca	b0	95	b9	80	f6
	f2	2b	43	49	89	f1	1a	3b	3b	89	f1	1a	f1	ac	a8	e5	f2	43	7a	7f
3	aa	61	82	68	ac	ef	13	45	ac	ef	13	45	75	20	53	bb	3d	47	1e	6d
	8f	dd	d2	32	73	c1	b5	23	c1	b5	23	73	ec	0b	c0	25	80	16	23	7a
	5f	e3	4a	46	cf	11	d6	5a	d6	5a	cf	11	09	63	cf	d0	47	fe	7e	88
	03	ef	d2	9a	7b	df	b5	b8	b8	7b	df	b5	93	33	7c	dc	7d	3e	44	3b
4	48	67	4d	d6	52	85	e3	f6	52	85	e3	f6	0f	60	6f	5e	ef	a8	b6	db
	6c	1d	e3	5f	50	a4	11	cf	a4	11	cf	50	d6	31	c0	b3	44	52	71	0b
	4e	9d	b1	58	2f	5e	c8	6a	c8	6a	2f	5e	da	38	10	13	a5	5b	25	ad
	ee	0d	38	e7	28	d7	07	94	94	28	d7	07	a9	bf	6b	01	41	7f	3b	00
5	e0	c8	d9	85	e1	e8	35	97	e1	e8	35	97	25	bd	b6	4c	d4	7c	ca	11
	92	63	b1	b8	4f	fb	c8	6c	fb	c8	6c	4f	d1	11	3a	4c	d1	83	f2	f9
	7f	63	35	be	d2	fb	96	ae	96	ae	d2	fb	a9	d1	33	c0	c6	9d	b8	15
	e8	c0	50	01	9b	ba	53	7c	7c	9b	ba	53	ad	68	8e	b0	f8	87	bc	bc

6	f1	c1	7c	5d	a1	78	10	4c	a1	78	10	4c	4b	2c	33	37	6d	11	db	ca
	00	92	c8	b5	63	4f	e8	d5	4f	e8	d5	63	86	4a	9d	d2	88	0b	f9	00
	6f	4c	8b	d5	a8	29	3d	03	3d	03	a8	29	8d	89	f4	18	a3	3e	86	93
	55	ef	32	0c	fc	df	23	fe	fe	fc	df	23	6d	80	e8	d8	7a	fd	41	fd
7	26	3d	e8	fd	f7	27	9b	54	f7	27	9b	54	14	46	27	34	4e	5f	84	4e
	0e	41	64	d2	ab	83	43	b5	83	43	b5	ab	15	16	46	2a	54	5f	a6	a6
	2e	b7	72	8b	31	a9	40	3d	40	3d	31	a9	b5	15	56	d8	f7	c9	4f	dc
	17	7d	a9	25	f0	ff	d3	3f	3f	f0	ff	d3	bf	ec	d7	43	0e	f3	b2	4f
8	5a	19	a3	7a	be	d4	0a	da	be	d4	0a	da	00	b1	54	fa	ea	b5	31	7f
	41	49	e0	8c	83	3b	e1	64	3b	e1	64	83	51	c8	76	1b	d2	8d	2b	8d
	42	dc	19	04	2c	86	d4	f2	d4	f2	2c	86	2f	89	6d	99	73	ba	f5	29
	b1	1f	65	0c	c8	c0	4d	fe	fe	c8	c0	4d	d1	ff	cd	ea	21	d2	60	2f
9	ea	04	65	85	87	f2	4d	97	87	f2	4d	97	47	40	a3	4c	ac	19	28	57
	83	45	5d	96	ec	6e	4c	90	6e	4c	90	ec	37	d4	70	9f	77	fa	d1	5c
	5c	33	98	b0	4a	c3	46	e7	46	e7	4a	c3	94	e4	3a	42	66	dc	29	00
	f0	2d	ad	c5	8c	d8	95	a6	a6	8c	d8	95	ed	a5	a6	bc	f3	21	41	6e
10	eb	59	8b	1b	e9	cb	3d	af	e9	cb	3d	af					d0	c9	e1	b6
	40	2e	a1	c3	09	31	32	2e	31	32	2e	09					14	ee	3f	63
	f2	38	13	42	89	07	7d	2c	7d	2c	89	07					f9	25	0c	0c
	1e	84	e7	d2	72	5f	94	b5	b5	72	5f	94					a8	89	c8	a6
output	39	02	dc	19																
	25	dc	11	6a																
	84	09	85	0b																
	1d	fb	97	32																

Appendix C — Example Vectors

The NIST Computer Security Resource Center provides a website with "examples with intermediate values" for AES [8].

Appendix D — Change Log (Informative)

The original FIPS 197 (November 26, 2001) was reviewed and updated under the auspices of NIST's Crypto Publication Review Board [9]. Public comments and analyses of the security of the AES that are described in NIST IR 8319 [10] were the basis for the decision to maintain the technical specifications of the Standard.

The following is a summary of the editorial changes to the original FIPS 197 in the May 9, 2023 update, NIST FIPS 197-upd1:

1. The formatting of many elements of the publication was improved, and the text was revised for clarity.
2. The following items were added to the front matter: title page, foreword, abstract, and keywords. Officials' names and affiliations on the title page reflect the original publication.
3. The announcement sections were updated to reflect current statutes, regulations, standards, guidelines, and validation programs.
4. Section 1 was revised to 1) add and update references to the AES development effort and 2) explicitly name AES-128, AES-192, and AES-256.
5. The material in the previous Section 2.2 (Algorithm Parameters, Symbols and Functions) was split into two new sections: 2.2 (List of Functions) and 2.3 (Algorithm Parameters and Symbols).
6. The terms, functions, and symbols from the specifications are comprehensively included in the lists in Sections 2.1–2.3.
7. The description of the indexing convention was removed from Section 3.1.
8. Table 1 was revised, and the text in the previous Section 3.2 on the polynomial interpretation of bytes was revised and moved to Section 4.
9. A general definition of the indexing of byte sequences was added to Section 3.3 before specializing to the example of a block, and Table 2 was revised.
10. The heading for Section 3.5 was changed to focus on word arrays, and notation for them was included in the text. The column words of the state were presented in a vertical format, with an improved description of the indices.
11. A reference for additional information on finite fields [4] was included in a footnote within Section 4, and the headings for Sections 4.1 and 4.2 were revised to explicitly mention GF(2^8).
12. Section 4.2 was revised to provide an explicit, general description of finite field multiplication. The previous Section 4.2.1 was incorporated into the revised Section 4.2 by replacing the original example of modular polynomial reduction with an illustration of finite field multiplication using `xtime`.
13. The heading of Section 4.3 was revised to focus on multiplication by a fixed matrix, and the text of the section was simplified by removing the secondary interpretation as polynomial

reduction. The descriptions of MIXCOLUMNS() and INVMIXCOLUMNS() in Sections 5.1.3 and 5.3.3 were revised accordingly, to refer back to this construction.

14. The text on multiplicative inverses in GF(2^8) from the previous Section 4.2 was revised and moved to the new Section 4.4.

15. The discussion of the algorithm specifications in Section 5 was expanded to elaborate on the relationships among its components. A new brief explanation of *Nb* as a Rijndael parameter enabled the replacement of *Nb* with its constant value 4 in the rest of the Standard.

16. The pseudocode for the cipher, the key expansion routine, and the inverse cipher in Sections 5.1, 5.2, and 5.3 was reformatted, and some of the text in these sections was revised for clarity.

17. The descriptions of SHIFTROWS() in Section 5.1.2 and INVSHIFTROWS() in Section 5.3.2 were improved, and a mistake in the latter was corrected.

18. Illustrations of the three instances of KEYEXPANSION() in the new Figs. 6, 7, and 8 were added to Section 5.2. The text in the section was also revised, including an explicit display of the round constants in the new Fig. 5.

19. A separate algorithm for the modified key expansion routine for the equivalent inverse cipher was added to Section 5.3.5 instead of only the supplementary lines. The description of the equivalent inverse cipher was simplified in favor of the citation of an updated reference [3].

20. Section 6.2 was revised to include a reference to NIST Special Publication 800-133, Rev. 2 [6].

21. Section 6.4 was revised to expand the discussion of implementation attacks.

22. The References section is no longer labeled as an appendix. The references were updated to replace withdrawn publications and correct citation information and URLs.

23. The examples in Appendix C were removed in favor of a reference to the detailed example vectors that are now maintained at [8].

24. Appendix D was created to summarize the changes in this update to FIPS 197.