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 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).


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.


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:

Federal Information
Processing Standards Publication 197

Specification for the
ADVANCED ENCRYPTION STANDARD (AES)

Table of Contents

1 Introduction .................................................. 1

2 Definitions ................................................... 2
   2.1 Terms and Acronyms ...................................... 2
   2.2 List of Functions ........................................ 3
   2.3 Algorithm Parameters and Symbols ....................... 4

3 Notation and Conventions .................................. 5
   3.1 Inputs and Outputs ...................................... 5
   3.2 Bytes .................................................. 5
   3.3 Indexing of Byte Sequences ............................... 5
   3.4 The State .............................................. 6
   3.5 Arrays of Words ....................................... 7

4 Mathematical Preliminaries ............................... 8
   4.1 Addition in GF(2^8) ...................................... 8
   4.2 Multiplication in GF(2^8) ................................ 8
   4.3 Multiplication of Words by a Fixed Matrix ............... 9
   4.4 Multiplicative Inverses in GF(2^8) ....................... 10

5 Algorithm Specifications ................................. 11
   5.1 CIPHER() .............................................. 12
      5.1.1 SUBBYTES() ....................................... 13
      5.1.2 SHIFTROWS() ...................................... 14
      5.1.3 MIXCOLUMNS() .................................... 15
      5.1.4 ADDROUNDKEY() .................................. 16
   5.2 KEYSPECIFICATION() .................................... 17
5.3 InvCipher() .......................................................... 18
  5.3.1 InvShiftRows() ................................................. 22
  5.3.2 InvSubBytes() .................................................. 23
  5.3.3 InvMixColumns() ............................................. 24
  5.3.4 Inverse of AddRoundKey() ................................. 24
  5.3.5 EqInvCipher() ............................................... 24

6 Implementation Considerations 26
  6.1 Key Length Requirements ........................................ 26
  6.2 Keying Restrictions ............................................... 26
  6.3 Parameter Extensions ............................................. 26
  6.4 Implementation Suggestions Regarding Various Platforms .... 26
  6.5 Modes of Operation ............................................... 27

References 28

Appendix A — Key Expansion Examples 29
  A.1 Expansion of a 128-bit Key ................................... 29
  A.2 Expansion of a 192-bit Key ................................... 30
  A.3 Expansion of a 256-bit Key ................................... 32

Appendix B — Cipher Example 34

Appendix C — Example Vectors 36

Appendix D — Change Log (Informative) 37
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>Hexadecimal representation of 4-bit sequences</td>
<td>5</td>
</tr>
<tr>
<td>Table 2</td>
<td>Indices for bytes and bits</td>
<td>6</td>
</tr>
<tr>
<td>Table 3</td>
<td>Key-Block-Round Combinations</td>
<td>11</td>
</tr>
<tr>
<td>Table 4</td>
<td>SBOX(): substitution values for the byte $xy$ (in hexadecimal format)</td>
<td>14</td>
</tr>
<tr>
<td>Table 5</td>
<td>Round constants</td>
<td>17</td>
</tr>
<tr>
<td>Table 6</td>
<td>INVSBOX(): substitution values for the byte $xy$ (in hexadecimal format)</td>
<td>23</td>
</tr>
</tbody>
</table>

List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>State array input and output</td>
<td>7</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Illustration of SUBBYTES()</td>
<td>13</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Illustration of SHIFTROWS()</td>
<td>15</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Illustration of MIXCOLUMNS()</td>
<td>16</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Illustration of ADDROUNDKEY()</td>
<td>16</td>
</tr>
<tr>
<td>Figure 6</td>
<td>KEYEXPANSION() of AES-128 to generate the words $w[i]$ for $4 \leq i &lt; 44$, where $l$ ranges over the multiples of 4 between 0 and 36</td>
<td>19</td>
</tr>
<tr>
<td>Figure 7</td>
<td>KEYEXPANSION() of AES-192 to generate the words $w[i]$ for $6 \leq i &lt; 52$, where $l$ ranges over the multiples of 6 between 0 and 42</td>
<td>20</td>
</tr>
<tr>
<td>Figure 8</td>
<td>KEYEXPANSION() of AES-256 to generate the words $w[i]$ for $8 \leq i &lt; 60$, where $l$ ranges over the multiples of 8 between 0 and 48</td>
<td>21</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Illustration of INVSHIFTROWS()</td>
<td>23</td>
</tr>
</tbody>
</table>

List of Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm 1</td>
<td>Pseudocode for CIPHER()</td>
<td>12</td>
</tr>
<tr>
<td>Algorithm 2</td>
<td>Pseudocode for KEYEXPANSION()</td>
<td>18</td>
</tr>
<tr>
<td>Algorithm 3</td>
<td>Pseudocode for INVCIPHER()</td>
<td>22</td>
</tr>
<tr>
<td>Algorithm 4</td>
<td>Pseudocode for EQINVCIPHER()</td>
<td>25</td>
</tr>
<tr>
<td>Algorithm 5</td>
<td>Pseudocode for KEYEXPANSIONEIC()</td>
<td>25</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard.</td>
</tr>
<tr>
<td>Affine transformation</td>
<td>A transformation consisting of multiplication by a matrix, followed by the addition of a vector.</td>
</tr>
<tr>
<td>Array</td>
<td>A fixed-size data structure that stores a collection of elements, where each element is identified by its integer index or indices.</td>
</tr>
<tr>
<td>Bit</td>
<td>A binary digit: 0 or 1.</td>
</tr>
<tr>
<td>Block</td>
<td>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.</td>
</tr>
<tr>
<td>Block cipher</td>
<td>A family of permutations of blocks that is parameterized by the key.</td>
</tr>
<tr>
<td>Byte</td>
<td>A sequence of eight bits.</td>
</tr>
<tr>
<td>Equivalent inverse cipher</td>
<td>An alternative specification of the inverse of CIPHER() with a structure similar to that of CIPHER() and with a modified key schedule as input.</td>
</tr>
<tr>
<td>Key</td>
<td>The parameter of a block cipher that determines the selection of a permutation from the block cipher family.</td>
</tr>
<tr>
<td>Key schedule</td>
<td>The sequence of round keys that are generated from the key by KEYEXPANSION().</td>
</tr>
<tr>
<td>Rijndael</td>
<td>The block cipher that NIST selected as the winner of the AES competition.</td>
</tr>
<tr>
<td>Round</td>
<td>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.</td>
</tr>
<tr>
<td>Round key</td>
<td>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.</td>
</tr>
<tr>
<td>State</td>
<td>Intermediate result of the AES block cipher that is represented as a two-dimensional array of bytes with four rows and ( Nb ) columns.</td>
</tr>
<tr>
<td>S-box</td>
<td>A non-linear substitution table used in SUBBYTES() and KEYEXPANSION() to perform a one-to-one substitution of a byte value.</td>
</tr>
<tr>
<td>Word</td>
<td>A group of 32 bits that is treated either as a single entity or as an array of 4 bytes.</td>
</tr>
</tbody>
</table>
2.2 List of Functions

The following functions are specified in this Standard:

- **ADD ROUND KEY()**: 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().
- **INVSHIFTROWS()**: The inverse of SHIFTROWS().
- **INVSUBBYTES()**: The inverse of SUBBYTES().
- **KEYEXPANSION()**: The routine that generates the round keys from the key.
- **KEYEXPANSION EIC()**: 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 \( \text{GF}(2^8) \).

\[ \tilde{b} \] 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.

\( \text{GF}(2) \) Finite field with two elements.

\( \text{GF}(2^8) \) Finite field with 256 elements.

\( \text{in} \) The data input to \( \text{CIPHER}() \) or \( \text{INVCIPHER}() \), 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 \( \text{GF}(2^8) \).

\( \text{key} \) 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.

\( \text{out} \) The data output of \( \text{CIPHER}() \) or \( \text{INVCIPHER}() \), represented as an array of 16 bytes indexed from 0 to 15.

\( Rcon \) Word array for the round constant.

\( \text{state} \) 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.

\( \cdot \) Multiplication in \( \text{GF}(2^8) \).

\( * \) 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 (e.g., \{10100011\}). 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 (i.e., \{b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0\}).

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 \{10100011\} is \{a3\}.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>0000</th>
<th>0001</th>
<th>0010</th>
<th>0011</th>
<th>0100</th>
<th>0101</th>
<th>0110</th>
<th>0111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sequence</th>
<th>1000</th>
<th>1001</th>
<th>1010</th>
<th>1011</th>
<th>1100</th>
<th>1101</th>
<th>1110</th>
<th>1111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>8</td>
<td>9</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
</tr>
</tbody>
</table>

### Table 1. Hexadecimal representation of 4-bit sequences

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,

\[
r_0\ r_1\ r_2\ \ldots\ r_{(8k-3)}\ r_{(8k-2)}\ r_{(8k-1)}
\]

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

\[
a_j = \{r_{8j}\ r_{(8j+1)}\ \ldots\ r_{(8j+7)}\}.
\]

Thus, for example, the data block

\[
r_0\ r_1\ r_2\ \ldots\ r_{125}\ r_{126}\ r_{127}
\]

(3.3)
is represented by the byte sequence

\[ a_0 \ a_1 \ a_2 \ \ldots \ a_{13} \ a_{14} \ a_{15}, \]  

(3.4)

where

\[ a_0 = \{ r_0 \ r_1 \ \ldots \ r_7 \}; \]
\[ a_1 = \{ r_8 \ r_9 \ \ldots \ r_{15} \}; \]
\[ \vdots \]
\[ a_{15} = \{ r_{120} \ r_{121} \ \ldots \ r_{127} \}. \]

(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.

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

### 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 \( 0 \leq r < 4 \) and a column index \( c \) in the range \( 0 \leq c < 4 \). 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 \( \text{in}_0, \text{in}_1, \ldots, \text{in}_{15} \) to the state array \( s \) as follows:

\[ s[r,c] = \text{in}[r + 4c] \quad \text{for} \quad 0 \leq r < 4 \quad \text{and} \quad 0 \leq c < 4. \]

(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 \( \text{out}_0, \text{out}_1, \ldots, \text{out}_{15} \) as follows:

\[ \text{out}[r + 4c] = s[r,c] \quad \text{for} \quad 0 \leq r < 4 \quad \text{and} \quad 0 \leq c < 4. \]

(3.7)

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

\[
\begin{align*}
\mathbf{v}_0 &= \begin{pmatrix} s_{0,0} \\ s_{1,0} \\ s_{2,0} \\ s_{3,0} \end{pmatrix}, & \mathbf{v}_1 &= \begin{pmatrix} s_{0,1} \\ s_{1,1} \\ s_{2,1} \\ s_{3,1} \end{pmatrix}, & \mathbf{v}_2 &= \begin{pmatrix} s_{0,2} \\ s_{1,2} \\ s_{2,2} \\ s_{3,2} \end{pmatrix}, & \mathbf{v}_3 &= \begin{pmatrix} s_{0,3} \\ s_{1,3} \\ s_{2,3} \\ s_{3,3} \end{pmatrix}.
\end{align*}
\] (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:

\[
b(x) = b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x + b_0.
\]  
(4.1)

For example, \{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:

\[
\begin{align*}
(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) &= x^7 + x^6 + x^4 + x^2 \quad \text{(polynomial)} \\
\{01010111\} \oplus \{10000011\} &= \{11010100\} \quad \text{(binary)} \\
\{57\} \oplus \{83\} &= \{d4\} \quad \text{(hexadecimal)}.
\end{align*}
\]
(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 GF(2^8). 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:

\[
m(x) = x^8 + x^4 + x^3 + x + 1.
\]  
(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 $b \cdot c$ is represented by the following modular reduction of their product as polynomials:

$$b(x)c(x) \mod m(x).$$  \hfill (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., $c = \{02\}$). In particular, the product $b \cdot \{02\}$ can be expressed as a function of $b$, denoted by $\text{XTIMES}(b)$, as follows:

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

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

$$\begin{align*}
\{57\} \cdot \{01\} &= \{57\} \\
\{57\} \cdot \{02\} &= \text{XTIMES}(\{57\}) = \{ae\} \\
\{57\} \cdot \{04\} &= \text{XTIMES}(\{ae\}) = \{47\} \\
\{57\} \cdot \{08\} &= \text{XTIMES}(\{47\}) = \{8e\} \\
\{57\} \cdot \{10\} &= \text{XTIMES}(\{8e\}) = \{07\} \\
\{57\} \cdot \{20\} &= \text{XTIMES}(\{07\}) = \{0e\} \\
\{57\} \cdot \{40\} &= \text{XTIMES}(\{0e\}) = \{1c\} \\
\{57\} \cdot \{80\} &= \text{XTIMES}(\{1c\}) = \{38\}.
\end{align*}$$  \hfill (4.6)

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

$$\begin{align*}
\{57\} \cdot \{13\} &= \{57\} \cdot (\{01\} \oplus \{02\} \oplus \{10\}) \\
&= \{57\} \oplus \{ae\} \oplus \{07\} \\
&= \{fe\}.
\end{align*}$$  \hfill (4.7)

### 4.3 Multiplication of Words by a Fixed Matrix

Two transformations – $\text{MIXCOLUMNS}()$ and $\text{INV\MIXCOLUMNS}()$ – 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:
\[
d_0 = (a_0 \cdot b_0) \oplus (a_3 \cdot b_1) \oplus (a_2 \cdot b_2) \oplus (a_1 \cdot b_3)
\]
\[
d_1 = (a_1 \cdot b_0) \oplus (a_0 \cdot b_1) \oplus (a_3 \cdot b_2) \oplus (a_2 \cdot b_3)
\]
\[
d_2 = (a_2 \cdot b_0) \oplus (a_1 \cdot b_1) \oplus (a_0 \cdot b_2) \oplus (a_3 \cdot b_3)
\]
\[
d_3 = (a_3 \cdot b_0) \oplus (a_2 \cdot b_1) \oplus (a_1 \cdot b_2) \oplus (a_0 \cdot b_3).
\]

The matrix form of Eq. (4.8) is

\[
\begin{bmatrix}
    d_0 \\
    d_1 \\
    d_2 \\
    d_3 \\
\end{bmatrix} =
\begin{bmatrix}
    a_0 & a_3 & a_2 & a_1 \\
    a_1 & a_0 & a_3 & a_2 \\
    a_2 & a_1 & a_0 & a_3 \\
    a_3 & a_2 & a_1 & a_0 \\
\end{bmatrix}
\begin{bmatrix}
    b_0 \\
    b_1 \\
    b_2 \\
    b_3 \\
\end{bmatrix}.
\] (4.9)

4.4 Multiplicative Inverses in GF(2^8)

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

\[
b \cdot b^{-1} = \{01\}.
\] (4.10)

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

\[
b^{-1} = b^{254}.
\] (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

\[
b(x)a(x) + m(x)c(x) = 1.
\] (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 \texttt{CIPHER()}; its inverse is denoted by \texttt{INVCIPHER()}\textsuperscript{2}.

The core of the algorithms for \texttt{CIPHER()} and \texttt{INVCIPHER()} is a sequence of fixed transformations of the state called a \textit{round}. Each round requires an additional input called the \textit{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 \texttt{KEYEXPANSION()}, takes the block cipher key as input and generates the round keys as output. In particular, the input to \texttt{KEYEXPANSION()} is represented as an array of words, denoted by \textit{key}, and the output is an expanded array of words, denoted by \textit{w}, called the \textit{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 \texttt{KEYEXPANSION()}. For each algorithm, the number of rounds is denoted by \textit{Nr}, and the number of words of the key is denoted by \textit{Nk}. (The number of words in the state is denoted by \textit{Nb} for Rijndael in general; in this Standard, \textit{Nb} = 4.) The specific values of \textit{Nk}, \textit{Nb}, and \textit{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>
<thead>
<tr>
<th>Key length</th>
<th>Block size</th>
<th>Number of rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 (in bits)</td>
<td>4 (in bits)</td>
<td>10</td>
</tr>
<tr>
<td>6 (in bits)</td>
<td>4 (in bits)</td>
<td>12</td>
</tr>
<tr>
<td>8 (in bits)</td>
<td>4 (in bits)</td>
<td>14</td>
</tr>
</tbody>
</table>

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

\begin{align*}
\text{AES-128}(\text{in}, \text{key}) &= \text{\texttt{CIPHER}}(\text{in}, 10, \text{\texttt{KEYEXPANSION}}(\text{key})) \\
\text{AES-192}(\text{in}, \text{key}) &= \text{\texttt{CIPHER}}(\text{in}, 12, \text{\texttt{KEYEXPANSION}}(\text{key})) \\
\text{AES-256}(\text{in}, \text{key}) &= \text{\texttt{CIPHER}}(\text{in}, 14, \text{\texttt{KEYEXPANSION}}(\text{key})).
\end{align*}

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

\textsuperscript{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 \(^0\) as a superscript on the original variable (i.e., by \(^b_i\), \(b^i\), \(s_{i,j}^0\), or \(s^i\)).

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 \(4 \times (Nr + 1)\) words.

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

```
Algorithm 1 Pseudocode for CIPHER()
1: procedure CIPHER(in, Nr, w)
2:    state ← in \Comment{See Sec. 3.4}
3:    state ← ADDRoundKey(state, w[0..3]) \Comment{See Sec. 5.1.4}
4:    for round from 1 to Nr - 1 do
5:        state ← SUBBYTES(state) \Comment{See Sec. 5.1.1}
6:        state ← SHIFTROWS(state) \Comment{See Sec. 5.1.2}
7:        state ← MIXCOLUMNS(state) \Comment{See Sec. 5.1.3}
8:        state ← ADDRoundKey(state, w[4*round..4*round + 3])
9:    end for
10:   state ← SUBBYTES(state) \Comment{See Sec. 3.4}
11:   state ← SHIFTROWS(state)
12:   state ← ADDRoundKey(state, w[4*Nr..4*Nr + 3])
13:   return state
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 $b' = SBOX(b)$ 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:

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

   \hspace{1cm} (5.2)

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

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

   \hspace{1cm} (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 & 1 & 1 \\
    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 \\
    0 \\
    0 \\
    0 \\
    1 \\
    1 \\
    1 \\
    0
\end{bmatrix}.  

(5.4)

Figure 2 illustrates how **SUBBYTES()** transforms the state.

![Figure 2. Illustration of SUBBYTES()](image)

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)

| $x$ | $0$ | $1$ | $2$ | $3$ | $4$ | $5$ | $6$ | $7$ | $8$ | $9$ | $a$ | $b$ | $c$ | $d$ | $e$ | $f$ |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| $0$ | 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 | 3a | 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 | le | 87 | e9 | ce | 55 | 28 | df |
| $f$ | 8c | al | 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 $s'_{r,c} = \{ed\}$.

### 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:

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

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.
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:

\[ [a_0, a_1, a_2, a_3] = [\{02\}, \{01\}, \{01\}, \{03\}] \]  

Thus,

\[
\begin{bmatrix}
\sigma'_{0,c} \\
\sigma'_{1,c} \\
\sigma'_{2,c} \\
\sigma'_{3,c}
\end{bmatrix}
= \begin{bmatrix}
02 & 03 & 01 & 01 \\
01 & 02 & 03 & 01 \\
01 & 01 & 02 & 03 \\
03 & 01 & 01 & 02
\end{bmatrix}
\begin{bmatrix}
\sigma_{0,c} \\
\sigma_{1,c} \\
\sigma_{2,c} \\
\sigma_{3,c}
\end{bmatrix}
\]

for \(0 \leq c < 4\),

\[
(5.7)
\]

so that the individual output bytes are defined as follows:

\[
\sigma'_{0,c} = (\{02\} \cdot \sigma_{0,c}) \oplus (\{03\} \cdot \sigma_{1,c}) \oplus \sigma_{2,c} \oplus \sigma_{3,c} \\
\sigma'_{1,c} = \sigma_{0,c} \oplus (\{02\} \cdot \sigma_{1,c}) \oplus (\{03\} \cdot \sigma_{2,c}) \oplus \sigma_{3,c} \\
\sigma'_{2,c} = \sigma_{0,c} \oplus \sigma_{1,c} \oplus (\{02\} \cdot \sigma_{2,c}) \oplus (\{03\} \cdot \sigma_{3,c}) \\
\sigma'_{3,c} = (\{03\} \cdot \sigma_{0,c}) \oplus \sigma_{1,c} \oplus \sigma_{2,c} \oplus (\{02\} \cdot \sigma_{3,c}).
\]

(5.8)

Figure 4 illustrates 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:

\[
\begin{bmatrix}
    s_{0,c} \\
    s_{1,c} \\
    s_{2,c} \\
    s_{3,c}
\end{bmatrix}
\oplus
\begin{bmatrix}
    s_{0,c}' \\
    s_{1,c}' \\
    s_{2,c}' \\
    s_{3,c}'
\end{bmatrix}
\]

where \( c \) is a value in the range \( 0 \leq \text{round} \leq N_r \), and \( \text{round} \) is a value in the range \( 0 \leq \text{round} \leq \text{Nr} \). In the specification of Cipher(), AddRoundKey() is invoked \( \text{Nr} + 1 \) times — once prior to the first application of the round function (see Alg. 1) and once within each of the \( \text{Nr} \) rounds, when \( 1 \leq \text{round} \leq \text{Nr} \).

The action of this transformation is illustrated in Fig. 5, where \( l = 4 \times \text{round} \). The byte address within words of the key schedule was described in Sec. 3.5.
5.2 KEYEXPANSION()

KEYEXPANSION() is a routine that is applied to the key to generate $4 \times (N_r + 1)$ words. Thus, four words are generated for each of the $N_r + 1$ 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 $0 \leq i < 4 \times (N_r + 1)$.

KEYEXPANSION() invokes 10 fixed words denoted by $Rcon[j]$ for $1 \leq j \leq 10$. 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>
<thead>
<tr>
<th>$j$</th>
<th>$Rcon[j]$</th>
<th>$j$</th>
<th>$Rcon[j]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[01, 00, 00, 00]</td>
<td>6</td>
<td>[20, 00, 00, 00]</td>
</tr>
<tr>
<td>2</td>
<td>[02, 00, 00, 00]</td>
<td>7</td>
<td>[40, 00, 00, 00]</td>
</tr>
<tr>
<td>3</td>
<td>[04, 00, 00, 00]</td>
<td>8</td>
<td>[80, 00, 00, 00]</td>
</tr>
<tr>
<td>4</td>
<td>[08, 00, 00, 00]</td>
<td>9</td>
<td>[1b, 00, 00, 00]</td>
</tr>
<tr>
<td>5</td>
<td>[10, 00, 00, 00]</td>
<td>10</td>
<td>[36, 00, 00, 00]</td>
</tr>
</tbody>
</table>

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 $\times T I M E S()$ 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,

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

and

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

The expansion of the key proceeds according to the pseudocode in Alg. 2. The first $N_k$ 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 $N_k$ positions earlier, $w[i-N_k]$, as follows:

- If $i$ is a multiple of $N_k$, then $w[i] = w[i-N_k] \oplus \text{SUBWORD} \left( \text{ROTWORD}(w[i-1]) \right) \oplus Rcon[i/N_k]$.
- For AES-256, if $i+4$ is a multiple of 8, then $w[i] = w[i-N_k] \oplus \text{SUBWORD}(w[i-1])$.
- For all other cases, $w[i] = w[i-N_k] \oplus w[i-1]$.
Algorithm 2 Pseudocode for KEYEXPANSION()  

1: procedure KEYEXPANSION(key) 
2: \hspace{1em} i ← 0 
3: \hspace{1em} while \( i \leq Nk - 1 \) do 
4: \hspace{2em} \( w[i] \leftarrow key[4 \times i..4 \times i + 3] \) 
5: \hspace{2em} i ← i + 1 
6: \hspace{1em} \end while \hspace{1em} \triangleright \text{When the loop concludes, } i = Nk. 
7: \hspace{1em} while \( i \leq 4 \times Nr + 3 \) do 
8: \hspace{2em} temp \leftarrow w[i - 1] 
9: \hspace{2em} if \( i \mod Nk = 0 \) then 
10: \hspace{3em} temp \leftarrow \text{SUBWORD} (\text{ROTWORD} (temp)) \oplus \text{Rcon}[i/Nk] 
11: \hspace{2em} else if \( Nk > 6 \) and \( i \mod Nk = 4 \) then 
12: \hspace{3em} temp \leftarrow \text{SUBWORD} (temp) 
13: \hspace{2em} \end if 
14: \hspace{2em} w[i] \leftarrow w[i - Nk] \oplus temp 
15: \hspace{2em} i \leftarrow i + 1 
16: \hspace{1em} \end while 
17: \hspace{1em} return w 
18: \hspace{1em} \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 \text{INVSHIFTROWS()}, \text{INVSUBBYTES()}, \text{INVMIXCOLUMNS()}, and \text{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 $8 \leq i < 60$, where $l$ ranges over the multiples of 8 between 0 and 48.
Algorithm 3 Pseudocode for \textsc{InvCipher()}

\begin{verbatim}
1: procedure \textsc{InvCipher}(in, Nr, w)
2:    state ← in \hfill \triangleright \text{ See Sec. 3.4}
3:    state ← \textsc{AddRoundKey}(state, w[4*Nr..4*Nr+3]) \hfill \triangleright \text{ See Sec. 5.1.4}
4:    \textbf{for} round from Nr - 1 \textbf{downto} 1 \textbf{do}
5:        state ← \textsc{InvShiftRows}(state) \hfill \triangleright \text{ See Sec. 5.3.1}
6:        state ← \textsc{InvSubBytes}(state) \hfill \triangleright \text{ See Sec. 5.3.2}
7:        state ← \textsc{AddRoundKey}(state, w[4*round..4*round+3])
8:        state ← \textsc{InvMixColumns}(state) \hfill \triangleright \text{ See Sec. 5.3.3}
9:    \textbf{end for}
10:   state ← \textsc{InvShiftRows}(state)
11:   state ← \textsc{InvSubBytes}(state)
12:   state ← \textsc{AddRoundKey}(state, w[0..3])
13: return state
14: \textbf{end procedure}
\end{verbatim}

5.3.1 \textsc{InvShiftRows()}

\textsc{InvShiftRows()} is the inverse of the \textsc{ShiftRows()}. In particular, the bytes in the last three rows of the state are cyclically shifted as follows:

\begin{equation}
    s'_{r,c} = s_{r,(c-r) \mod 4} \quad \text{for } 0 \leq r < 4 \text{ and } 0 \leq c < 4. \tag{5.12}
\end{equation}

\textsc{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.
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:

| x  | 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 | 6b | 2b | 9d | 24 | b2 | 76 | 5b | a2 | 49 | 6d | 8b | d1 | 25 |
| 4  | 72 | f8 | f6 | 64 | 86 | 68 | 98 | 16 | d4 | a4 | 5c | cc | 5d | 6b | 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 | 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 | la | 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:

\[ [a_0, a_1, a_2, a_3] = \{\{0e\}, \{09\}, \{0d\}, \{0b\}] \]

Thus,

\[
\begin{bmatrix}
  s'_{0,c} \\
  s'_{1,c} \\
  s'_{2,c} \\
  s'_{3,c}
\end{bmatrix} = \begin{bmatrix}
  0e & 0b & 0d & 09 \\
  09 & 0e & 0b & 0d \\
  0d & 09 & 0e & 0b \\
  0b & 0d & 09 & 0e
\end{bmatrix} \begin{bmatrix}
  s_{0,c} \\
  s_{1,c} \\
  s_{2,c} \\
  s_{3,c}
\end{bmatrix}
\text{ for } 0 \leq c < 4.
\]

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

\[
\begin{align*}
  s'_{0,c} &= (\{0e\} \cdot s_{0,c}) \oplus (\{0b\} \cdot s_{1,c}) \oplus (\{0d\} \cdot s_{2,c}) \oplus (\{09\} \cdot s_{3,c}) \\
  s'_{1,c} &= (\{09\} \cdot s_{0,c}) \oplus (\{0e\} \cdot s_{1,c}) \oplus (\{0d\} \cdot s_{2,c}) \oplus (\{0b\} \cdot s_{3,c}) \\
  s'_{2,c} &= (\{0d\} \cdot s_{0,c}) \oplus (\{09\} \cdot s_{1,c}) \oplus (\{0b\} \cdot s_{2,c}) \oplus (\{0e\} \cdot s_{3,c}) \\
  s'_{3,c} &= (\{0b\} \cdot s_{0,c}) \oplus (\{0d\} \cdot s_{1,c}) \oplus (\{09\} \cdot s_{2,c}) \oplus (\{0e\} \cdot s_{3,c})
\end{align*}
\]

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()

1: procedure EQINVCIPHER(in, Nr, dw)
2:     state ← in
3:     state ← ADDROUNDKEY(state, dw[4*Nr..4*Nr + 3])
4: for round from Nr – 1 downto 1 do
5:     state ← INVSUBBYTES(state)
6:     state ← INVSHIFTROWS(state)
7:     state ← INVMIXCOLUMNS(state)
8:     state ← ADDROUNDKEY(state, dw[4*round..4*round + 3])
9: end for
10: state ← INVSUBBYTES(state)
11: state ← INVSHIFTROWS(state)
12: state ← ADDROUNDKEY(state, dw[0..3])
13: return state
14: 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., \(N_k = 4, 6, \text{ or } 8\), 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 \((N_k)\), block size \((N_b)\), and number of rounds \((N_r)\). 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


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:

\[
\text{Key} = 2b\ 7e\ 15\ 16\ 28\ \text{ae}\ d2\ a6\ \text{ab}\ f7\ 15\ 88\ 09\ \text{cf}\ 4f\ 3c
\]

for \( Nk = 4 \), which results in

\[
w_0 = 2b7e1516 \quad w_1 = 28aed2a6 \quad w_2 = \text{abf71588} \quad w_3 = 09cf4f3c
\]

<table>
<thead>
<tr>
<th>( i ) (dec)</th>
<th>( temp )</th>
<th>After ( \text{ROTWord()} )</th>
<th>After ( \text{SUBWord()} )</th>
<th>( Rcon[i/Nk] )</th>
<th>After ( \text{XOR} ) with ( Rcon )</th>
<th>( w[i-Nk] )</th>
<th>( w[i] = temp \oplus w[i-Nk] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>09cf4f3c</td>
<td>cf4f3c09</td>
<td>8a84eb01</td>
<td>01000000</td>
<td>8b84eb01</td>
<td>2b7e1516</td>
<td>a0faef17</td>
</tr>
<tr>
<td>5</td>
<td>a0faef17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>28aed2a6</td>
<td>88542cb1</td>
</tr>
<tr>
<td>6</td>
<td>88542cb1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>abf71588</td>
<td>23a33939</td>
</tr>
<tr>
<td>7</td>
<td>23a33939</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>09cf4f3c</td>
<td>2a6c7605</td>
</tr>
<tr>
<td>8</td>
<td>2a6c7605</td>
<td>6c76052a</td>
<td>50386be5</td>
<td>02000000</td>
<td>52386be5</td>
<td>a0faef17</td>
<td>f2c295f2</td>
</tr>
<tr>
<td>9</td>
<td>f2c295f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>88542cb1</td>
<td>7a96b943</td>
</tr>
<tr>
<td>10</td>
<td>7a96b943</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>23a33939</td>
<td>5935807a</td>
</tr>
<tr>
<td>11</td>
<td>5935807a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2a6c7605</td>
<td>7359f67f</td>
</tr>
<tr>
<td>12</td>
<td>7359f67f</td>
<td>59f67f73</td>
<td>cb42d28f</td>
<td>04000000</td>
<td>cf42d28f</td>
<td>f2c295f2</td>
<td>3d80477d</td>
</tr>
<tr>
<td>13</td>
<td>3d80477d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7a96b943</td>
<td>4716fe3e</td>
</tr>
<tr>
<td>14</td>
<td>4716fe3e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5935807a</td>
<td>1e237e44</td>
</tr>
<tr>
<td>15</td>
<td>1e237e44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7359f67f</td>
<td>6d7a883b</td>
</tr>
<tr>
<td>16</td>
<td>6d7a883b</td>
<td>7a883b6d</td>
<td>dac4e23c</td>
<td>08000000</td>
<td>d2c4e23c</td>
<td>3d80477d</td>
<td>ef44a541</td>
</tr>
<tr>
<td>17</td>
<td>ef44a541</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4716fe3e</td>
<td>a8525b7f</td>
</tr>
<tr>
<td>18</td>
<td>a8525b7f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1e237e44</td>
<td>b671253b</td>
</tr>
<tr>
<td>19</td>
<td>b671253b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6d7a883b</td>
<td>db0bad00</td>
</tr>
<tr>
<td>20</td>
<td>db0bad00</td>
<td>0bad00db</td>
<td>2b9563b9</td>
<td>10000000</td>
<td>3b9563b9</td>
<td>ef44a541</td>
<td>d4dlcf68</td>
</tr>
<tr>
<td>21</td>
<td>d4dlcf68</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a8525b7f</td>
<td>7c839d87</td>
</tr>
<tr>
<td>22</td>
<td>7c839d87</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>b671253b</td>
<td>caf2b8bc</td>
</tr>
<tr>
<td>23</td>
<td>caf2b8bc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>db0bad00</td>
<td>11f915bc</td>
</tr>
</tbody>
</table>
A.2 Expansion of a 192-bit Key

This section contains the key expansion of the following key:

\[
\text{Key} = \begin{array}{cccccccc}
8e & 73 & b0 & f7 & da & 0e & 64 & 52 \\
80 & 90 & 79 & e5 & 62 & f8 & ea & d2 \\
\end{array}
\]

for \(Nk = 6\), which results in

\[
\begin{align*}
w_0 &= 8e73b0f7 \\
w_1 &= da0e6452 \\
w_2 &= c810f32b \\
w_3 &= 809079e5 \\
w_4 &= 62f8ead2 \\
w_5 &= 522c6b7b
\end{align*}
\]

<table>
<thead>
<tr>
<th>(i)</th>
<th>(\text{temp})</th>
<th>(\text{After ROTWORD()})</th>
<th>(\text{After SubWord()})</th>
<th>(Rcon[i/Nk])</th>
<th>(\text{After XOR with } Rcon)</th>
<th>(w[i−Nk])</th>
<th>(w[i] = \text{temp} \oplus w[i−Nk])</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>522c6b7b</td>
<td>2c6b7b52</td>
<td>717f2100</td>
<td>01000000</td>
<td>707f2100</td>
<td>8e73b0f7</td>
<td>fe0c91f7</td>
</tr>
<tr>
<td>7</td>
<td>fe0c91f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>da0e6452</td>
<td>2402f5a5</td>
</tr>
<tr>
<td>8</td>
<td>2402f5a5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>c810f32b</td>
<td>ec12068e</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>eclado06e</td>
<td>809079e5 6c827f6b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>6c827f6b</td>
<td>62f8ead2 0e7a95b9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0e7a95b9</td>
<td>5226c6b7b 5c56f2c2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>5c56hec2 56hec25c b1bb254a 02000000 b3bb254a</td>
<td>fe0c91f7 4db7b4bd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>4d7b7b4bd</td>
<td>2402f5a5 69b54118</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>69c54118</td>
<td>ec1206d8 85a74796</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>85a74796</td>
<td>6c827f6b e92538fd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>e92538fd</td>
<td>0e7a95b9 e75f2d44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>e75f2d44</td>
<td>5c56f2c2 b095386</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>bb095386 095386bb 01ed44ea 04000000 05ed44ea</td>
<td>4db7b4bd 485af057</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>485af057</td>
<td>69b54118 21efb14f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>21efb14f</td>
<td>85a74796 a448f6d9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>a448f6d9</td>
<td>e92538fd 4d6dce24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>4d6dce24</td>
<td>e75f2d44 a326360</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>a326360</td>
<td>b095386 113b30e6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>113b30e6 3b30e611 e2048e82 08000000 ea048e82</td>
<td>485af057 a25e7ed5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>a25e7ed5</td>
<td>21efb14f 83b1c9f9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>83b1c9f9</td>
<td>a448f6d9 27f93943</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>27f93943</td>
<td>4d6dce24 6a94f767</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>6a94f767</td>
<td>aa326360 c0a69407</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>c0a69407</td>
<td>113b30e6 d19da4e1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>d19da4e1 9da4e1d1 5e49f83e 10000000 4e49f83e</td>
<td>a25e7ed5 ec1786eb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>ec1786eb</td>
<td>83b1c9f9 a6fa64971</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>6fa64971</td>
<td>27f93943 485f7032</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>485f7032</td>
<td>6a94f767 22cb8755</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>22cb8755</td>
<td>c0a69407 e26d1352</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>e26d1352</td>
<td>d19da4e1 33f0b7b3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>33f0b7b3 f0b7b333 8ca96dc3 20000000 aca96dc3</td>
<td>ec1786eb 40beeb28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>40beeb28</td>
<td>6fa64971 2f18a259</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>2f18a259</td>
<td>485f7032 6747d26b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>6747d26b</td>
<td>22cb8755 458c553e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>458c553e</td>
<td>e26d1352 a7e1466c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>a7e1466c</td>
<td>33f0b7b3 9411f1df</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>9411f1df 11f1df94 82a19e22 40000000 c2a19e22</td>
<td>40beeb28 821f750a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>821f750a</td>
<td>2f18a259 a07d753</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

\[
\begin{align*}
\text{w}_0 &= 603deb10 \\
\text{w}_1 &= 15ca71be \\
\text{w}_2 &= 2b73aef0 \\
\text{w}_3 &= 857d7781 \\
\text{w}_4 &= 1f352c07 \\
\text{w}_5 &= 3b6108d7 \\
\text{w}_6 &= 2d9810a3 \\
\text{w}_7 &= 0914dff4
\end{align*}
\]

<table>
<thead>
<tr>
<th>( i ) (dec)</th>
<th>( \text{temp} )</th>
<th>After \text{RO}T\text{Word}()</th>
<th>After \text{SU}b\text{WORD}()</th>
<th>( R\text{con}[i/Nk] )</th>
<th>After XOR with ( R\text{con} )</th>
<th>( w[i-Nk] )</th>
<th>( w[i] = \text{temp} \oplus w[i-Nk] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0914dff4</td>
<td>14dff409</td>
<td>fa9ebf01</td>
<td>01000000</td>
<td>fb9ebf01</td>
<td>603deb10</td>
<td>9ba35411</td>
</tr>
<tr>
<td>9</td>
<td>9ba35411</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15ca71be 8e6925af</td>
</tr>
<tr>
<td>10</td>
<td>8e6925af</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2b73aef0 a51a8b5f</td>
</tr>
<tr>
<td>11</td>
<td>a51a8b5f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>857d7781</td>
<td>2067fcde</td>
</tr>
<tr>
<td>12</td>
<td>2067fcde</td>
<td>b785b01d</td>
<td></td>
<td></td>
<td></td>
<td>1f352c07</td>
<td>a8b09c1a</td>
</tr>
<tr>
<td>13</td>
<td>a8b09c1a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3b6108d7</td>
<td>93d194cd</td>
</tr>
<tr>
<td>14</td>
<td>93d194cd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2d9810a3</td>
<td>be49846e</td>
</tr>
<tr>
<td>15</td>
<td>be49846e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0914dff4</td>
<td>b75d5b9a</td>
</tr>
<tr>
<td>16</td>
<td>b75d5b9a 5d5b9ab7</td>
<td>4c39b8a9</td>
<td>02000000</td>
<td>4e39b8a9</td>
<td>9ba35411</td>
<td>5bf3c917</td>
<td>a51a8b5f ffe94248</td>
</tr>
<tr>
<td>17</td>
<td>d59aecb8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8e6925af</td>
<td>5bf3c917 a51a8b5f ffe94248</td>
</tr>
<tr>
<td>18</td>
<td>5bf3c917</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>93d194cd</td>
<td>2678a647</td>
</tr>
<tr>
<td>19</td>
<td>fee94248</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2067fcde</td>
<td>de8ebe96</td>
</tr>
<tr>
<td>20</td>
<td>de8ebe96</td>
<td>1d19ae90</td>
<td></td>
<td></td>
<td></td>
<td>a8b09c1a b5a9328a</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>b5a9328a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>93d194cd</td>
<td>2678a647</td>
</tr>
<tr>
<td>22</td>
<td>2678a647</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>be49846e</td>
<td>98312229</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>98312229</td>
<td></td>
<td></td>
<td></td>
<td>b75d5b9a</td>
<td>2f6c79b3</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>2f6c79b3</td>
<td>6c79b32f</td>
<td>50b66d15</td>
<td>04000000</td>
<td>54b66d15</td>
<td>d59aecb8</td>
<td>812c81ad</td>
</tr>
<tr>
<td>25</td>
<td>812c81ad</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5bf3c917</td>
<td>dadf48ba</td>
</tr>
<tr>
<td>26</td>
<td>dadf48ba</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fef94248</td>
<td>24360af2</td>
</tr>
<tr>
<td>27</td>
<td>24360af2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>de8ebe96</td>
<td>fab8b464</td>
</tr>
<tr>
<td>28</td>
<td>fab8b464</td>
<td>2d6c8d43</td>
<td></td>
<td></td>
<td></td>
<td>b5a9328a</td>
<td>98c5bfc9</td>
</tr>
<tr>
<td>29</td>
<td>98c5bfc9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2678a647</td>
<td>bebd198e</td>
</tr>
<tr>
<td>30</td>
<td>bebd198e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>98312229</td>
<td>268c3ba7</td>
</tr>
<tr>
<td>31</td>
<td>268c3ba7</td>
<td></td>
<td></td>
<td>2f6c79b3</td>
<td>09e04214</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>09e04214</td>
<td>e0421409</td>
<td>e12cfa01</td>
<td>08000000</td>
<td>e92cfa01</td>
<td>812c81ad</td>
<td>68007bac</td>
</tr>
<tr>
<td>33</td>
<td>68007bac</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dadf48ba</td>
<td>b2df3316</td>
</tr>
<tr>
<td>34</td>
<td>b2df3316</td>
<td></td>
<td></td>
<td>24360af2</td>
<td>96e939e4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>96e939e4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fab8b464</td>
<td>6c518d80</td>
</tr>
<tr>
<td>36</td>
<td>6c518d80</td>
<td>50d15dcd</td>
<td></td>
<td></td>
<td></td>
<td>98c5bfc9</td>
<td>c814e204</td>
</tr>
<tr>
<td>37</td>
<td>c814e204</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bebd198e</td>
<td>76a9fb8a</td>
</tr>
<tr>
<td>38</td>
<td>76a9fb8a</td>
<td></td>
<td></td>
<td>268c3ba7</td>
<td>5025c02d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>5025c02d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>09e04214</td>
<td>59c58239</td>
</tr>
<tr>
<td>40</td>
<td>59c58239</td>
<td>c5823959</td>
<td>a61312cb</td>
<td>10000000</td>
<td>b61312cb</td>
<td>68007bac</td>
<td>d136967</td>
</tr>
<tr>
<td>41</td>
<td>de136967</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>b2df3316</td>
<td>6ccc5a71</td>
</tr>
<tr>
<td>42</td>
<td>6ccc5a71</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>96e939e4</td>
<td>fa256395</td>
</tr>
<tr>
<td>43</td>
<td>fa256395</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6c518d80</td>
<td>9674ee15</td>
</tr>
<tr>
<td>44</td>
<td>9674ee15</td>
<td>90922859</td>
<td></td>
<td></td>
<td></td>
<td>c814e204</td>
<td>5886ca5d</td>
</tr>
<tr>
<td>45</td>
<td>5886ca5d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>76a9fb8a</td>
<td>2e2f31d7</td>
</tr>
<tr>
<td>46</td>
<td>2e2f31d7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5025c02d</td>
<td>7e0af1fa</td>
</tr>
<tr>
<td>47</td>
<td>7e0af1fa</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>59c58239</td>
<td>27cf73c3</td>
</tr>
<tr>
<td>48</td>
<td>27cf73c3</td>
<td>cf73c327</td>
<td>8a8f2ecc</td>
<td>20000000</td>
<td>aa8f2ecc</td>
<td>de136967</td>
<td>749c47ab</td>
</tr>
<tr>
<td>49</td>
<td>749c47ab</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6ccc5a71</td>
<td>18501dda</td>
</tr>
<tr>
<td>50</td>
<td>18501dda</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fa256395</td>
<td>e2757e4f</td>
</tr>
<tr>
<td>51</td>
<td>e2757e4f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9674ee15</td>
<td>7401905a</td>
</tr>
<tr>
<td>52</td>
<td>7401905a</td>
<td>927c60be</td>
<td></td>
<td></td>
<td></td>
<td>5886ca5d</td>
<td>cafaaae3</td>
</tr>
<tr>
<td>53</td>
<td>cafaaae3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2e2f31d7</td>
<td>4d59b34</td>
</tr>
<tr>
<td>54</td>
<td>e4d59b34</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7e0af1fa</td>
<td>9adf6ace</td>
</tr>
<tr>
<td>55</td>
<td>9adf6ace</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>27cf73c3</td>
<td>bd10190d</td>
</tr>
<tr>
<td>56</td>
<td>bd10190d</td>
<td>10190dbd</td>
<td>cad4d77a</td>
<td>40000000</td>
<td>8ad4d77a</td>
<td>749c47ab</td>
<td>fe4890d1</td>
</tr>
<tr>
<td>57</td>
<td>fe4890d1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>18501dda</td>
<td>e6188d0b</td>
</tr>
<tr>
<td>58</td>
<td>e6188d0b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>e2757e4f</td>
<td>046df344</td>
</tr>
<tr>
<td>59</td>
<td>046df344</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7401905a</td>
<td>706c631e</td>
</tr>
</tbody>
</table>
### 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 3a 30 4e 3f 07 34

Key $= 2b$ 7e 15 16 28 ae d2 6a 6b f7 15 88 09 cf 4f 3c

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

<table>
<thead>
<tr>
<th>Round Number</th>
<th>Start of Round</th>
<th>SubBytes</th>
<th>ShiftRows</th>
<th>MixColumns</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>88 31</td>
<td>e0</td>
<td></td>
<td>2b 28 ab 09</td>
</tr>
<tr>
<td>2</td>
<td>9c</td>
<td>9f 5b</td>
<td>6a</td>
<td></td>
<td>7e ae f7 cf</td>
</tr>
<tr>
<td>3</td>
<td>5f</td>
<td>3e 4a</td>
<td>46</td>
<td></td>
<td>15 d2 15 4f</td>
</tr>
<tr>
<td>4</td>
<td>48</td>
<td>67 4d</td>
<td>d6</td>
<td></td>
<td>16 a6 88 3c</td>
</tr>
<tr>
<td>5</td>
<td>92</td>
<td>63 b1</td>
<td>8b</td>
<td></td>
<td>16 a6 88 3c</td>
</tr>
</tbody>
</table>
Tuesday, July 22, 2014

The Advanced Encryption Standard (AES), approved by NIST, is the FIPS 197. This standard is described in terms of its data processing and key schedule algorithms, and also includes a definition of the encryption algorithm and a description of its performance characteristics. The specification of the algorithm is in FIPS 197, which is available by request from NIST or from the NIST web site. The algorithm is described here in terms of data blocks of 128 bits and keys of 128, 192, or 256 bits.

The AES algorithm employs an advanced block cipher. It is an iterated, product cipher structure. It consists of a number of identical rounds performed on a data block. The FIPS 197 standard specifies the number of rounds and defines the operations of each round.

The encryption algorithm consists of a key expansion operation and 10, 12, or 14 rounds, depending on the key size.

The key expansion operation takes as input an input key and produces a series of subkeys used in the encryption algorithm.

The key expansion operation is performed on a 128-bit block of data. The input key is expanded into a number of 128-bit subkeys, where the number of subkeys is determined by the key size.

The encryption algorithm consists of 10, 12, or 14 rounds, depending on the key size.

Each round consists of a set of operations, including:

- AddRoundKey:
  - the input data block and the current subkey are combined using the XOR operation.
- SubBytes:
  - each byte of the input block is replaced with another byte according to a fixed substitution table.
- ShiftRows:
  - each row of the input block is shifted cyclically to the left.
- MixColumns:
  - each column of the input block is multiplied with a fixed matrix.

The output of the encryption algorithm is the encrypted data block.
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 \texttt{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 multiplication.
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 $\text{GF}(2^8)$ 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.