A Specification for Rijndael, the AES Algorithm

1. Notation and Conventions

1.1 Rijndael Inputs and Outputs

The input, output and cipher key for Rijndael are sequences containing 128, 160, 192, 224 or 256 bits with the input and output sequences having the same length (the cipher block size). A bit is a binary digit (0 or 1) and ‘length’ refers to the number of elements (in this case bits) in a sequence. In general the block size and the key length can be any of the five allowed values but for the Advanced Encryption Standard (AES) the block size is fixed at 128 bits and the key length can only be 128, 192 or 256 bits.

The elements within sequences and sub-sequences will be enumerated from zero to one less than the number of elements. The number associated with a sequence element is called its index and sequences will be presented with their lower numbered elements to the left. Unless otherwise specified, the enumeration order of sub-sequences matches that of the sequence from which they are derived. For the input, output and key sequences used in Rijndael, the number \( i \) associated with a bit will hence be in one of the five ranges \( 0 \leq i < 128 \), \( 0 \leq i < 160 \), \( 0 \leq i < 192 \), \( 0 \leq i < 224 \) or \( 0 \leq i < 256 \).

The mapping of bit sequences onto logical or physical resources is outside the scope of this document. But for software the preferred mapping, one used here in pseudo code, is onto arrays of 8-bit bytes with consecutive 8-bit sub-sequences forming consecutive bytes within which bits with lower bit sequence indexes have higher numeric significance.

1.2 Bytes

Internally bit sequences are interpreted as one-dimensional arrays of 8-bit bytes where byte \( n \) consists of the sub-sequence \( 8n \) to \( 8n + 7 \). In such arrays, denoted by \( a \), the \( n \)'th byte will be referred to as \( a_n \) or \( a[n] \), where \( n \) is in one of the ranges \( 0 \leq n < 16 \), \( 0 \leq n < 20 \), \( 0 \leq n < 24 \), \( 0 \leq n < 28 \) or \( 0 \leq n < 32 \). The order \( i \) of a bit within a byte has the value \( 7 - k \), where \( k \) is the bit’s index, and the bit with order \( i \) in a byte \( b \) will be denoted by \( b_i \). Internally bytes are polynomial representations of finite field elements:

\[
b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 = \sum_{i=0}^{7} b_i x^i \quad (1.2.1)
\]

The values of bytes will be presented as a concatenation of their bits between braces with higher order (lower index) bits to the left. Hence \{11000011\} identifies a specific finite field element. It is also convenient to denote byte values using hexadecimal notation, with each of two groups of four bits being denoted by a character as follows.

<table>
<thead>
<tr>
<th>bit pattern</th>
<th>character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>a</td>
</tr>
<tr>
<td>1011</td>
<td>b</td>
</tr>
<tr>
<td>1100</td>
<td>c</td>
</tr>
<tr>
<td>1101</td>
<td>d</td>
</tr>
<tr>
<td>1110</td>
<td>e</td>
</tr>
<tr>
<td>1111</td>
<td>f</td>
</tr>
</tbody>
</table>

Hence the value \{11000011\} can also be written as \{63\}, where the character denoting the 4-bit group containing the higher order bits is again to the left. Some finite field operations use an extra bit (\( b_8 \)) to the left of an 8-bit byte; if present, this will appear to the left of the left brace as in \{1\{1b\}\}. The external integer byte value 0x01 is mapped by the interface without translation or conversion to the internal finite field value \{01\}.

1.3 The Rijndael State

Internally Rijndael operates on a two dimensional array of bytes called the state that contains 4 rows and \( N_c \) columns, where \( N_c \) is the input sequence length divided by 32. In
this state array, denoted by the symbol $s$, each individual byte has two indexes: its row number $r$, in the range $0 \leq r < 4$, and its column number $c$, in the range $0 \leq c < Nc$, hence allowing it to be referred to either as $s_{r,c}$ or $s[r,c]$. For AES the range for $c$ is $0 \leq c < 4$ since $Nc$ has a fixed value of 4.

At the start (end) of an encryption or decryption operation the bytes of the cipher input (output) are copied to (from) this state array in the order shown in Figure 1.

![Figure 1](image)

**Figure 1 – Input to, and output from, the cipher state array**

Hence at the start (end) of encryption or decryption the input (output) array in (out) is copied to (from) the state array according to the schemes:

$$
\begin{align*}
    s[r,c] &= \text{in}[r + 4c] \\
    \text{out}[r + 4c] &= s[r,c]
\end{align*}

(1.3.1)$$

### 1.4 Arrays of 32-bit Words

The four bytes in each column of the state can be thought of as an array of four bytes indexed by the row number $r$ or as a single 32-bit word (bytes within all 32-bit words will always be enumerated using the index $r$). The state can thus be considered as a one-dimensional array of words for which the column number $c$ provides the array index.

The key schedule for Rijndael (described in Section 4) is an array of 32-bit words, denoted by the symbol $k$, with the lower elements initialised from the cipher key input so that byte $4i + r$ of the key is copied into byte $r$ of key schedule word $k[i]$. The cipher iterates through a number of cycles, called rounds, each of which uses $Nc$ words from this key schedule. This key schedule can also be viewed as an array of round keys, each of which consists of an $Nc$ word sub-array so that word $c$ of round key $n$, $k[Nc*n+c]$, can also be referred to using two dimensional array notation as either $k[n,c]$ or $k_{n,c}$. Here the round key for round $n$ as a whole, an $Nc$ word sub-array, will sometimes be referred to by replacing the second index with "$^*$" as in $k[n,*]$ and $k_{n,*}$.

### 2. Finite Field Operations

#### 2.1 Finite Field Addition

The addition of two finite field elements is achieved by adding the coefficients for corresponding powers in their polynomial representations, this addition being performed in GF(2), that is, modulo 2, so that $1 + 1 = 0$. Consequently, addition and subtraction are both equivalent to an exclusive-or operation on the bytes that represent field elements. Addition operations for finite field elements will be denoted by the symbol $\oplus$. For example, the following expressions are equivalent:

$$
\begin{align*}
    (x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) &\equiv (x^7 + x^6 + x^4 + x^2) & \text{(Polynomial)} \\
    \{01010111\} \oplus \{10000011\} &\equiv \{11010100\} & \text{(Binary)} \\
    \{57\} \oplus \{83\} &\equiv \{d4\} & \text{(Hexadecimal)}
\end{align*}
$$
2.2 Finite Field Multiplication

Finite field multiplication is more difficult than addition and is achieved by multiplying the polynomials for the two elements concerned and collecting like powers of \( x \) in the result. Since each polynomial can have powers of \( x \) up to 7, the result can have powers of \( x \) up to 14 and will no longer fit within a single byte.

This situation is handled by replacing the result with the remainder polynomial after division by a special eighth order irreducible polynomial, which, for Rijndael, is:

\[
m(x) = x^8 + x^4 + x^3 + x + 1
\]

Since this polynomial has powers of \( x \) up to 8 it cannot be represented by a single byte and will be written as either \( \text{1}\{00011011\} \) or \( \text{1}\{1b\} \) as indicated earlier. This process is illustrated in the following example of the product \( \{57\} \bullet \{83\} \equiv \{c1\} \) (where \( \bullet \) is used to represent finite field multiplication):

\[
(x^6 + x^4 + x^2 + x + 1) \bullet (x^7 + x + 1) \rightarrow
\]

\[
(x^6 + x^4 + x^2 + x + 1) \bullet x^7 = x^{13} + x^{11} + x^9 + x^8 + x^7
\]

\[
(x^6 + x^4 + x^2 + x + 1) \bullet x = x^7 + x^5 + x^3 + x^2 + x
\]

\[
(x^6 + x^4 + x^2 + x + 1) \bullet 1 = x^6 + x^4 + x^2 + x + 1
\]

This intermediate result is now divided by \( m(x) \) above:

\[
(x^8 + x^4 + x^3 + x + 1) \bullet x^5 =
\]

\[
x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1
\]

\[
x^{13} + x^9 + x^8 + x^6 + x^5 + 1
\]

\[
x^{11} + x^7 + x^6 + x^4 + x^3 + 1
\]

\[
x^7 + x^6 + 1
\]

Multiplication is associative, and there is a neutral element \( \{01\} \); for any binary polynomial \( b(x) \) of degree less than 8, the extended Euclidean algorithm can be used to compute polynomials \( a(x) \) and \( c(x) \) such that:

\[
b(x) \bullet a(x) \oplus m(x) \bullet c(x) = 1
\]

\[
b(x) \bullet a(x) \mod m(x) = 1
\]

which shows that the polynomials \( a(x) \) and \( b(x) \) are mutual inverses. Furthermore:

\[
a(x) \bullet (b(x) \oplus c(x)) = a(x) \bullet b(x) \oplus a(x) \bullet c(x)
\]

(2.2.3)

It hence follows that the set of 256 byte values, with the XOR as addition and multiplication as defined above has the structure of the finite field \( \text{GF}(256) \).

2.3 Multiplication by Repeated Shifts

The finite field element \( \{00000001\} \) is the representation of the polynomial \( x \), which means that multiplying another element by this value increases all its powers of \( x \) by 1. This is equivalent to shifting its byte representation up by one bit so that the bit at position \( i \) moves to position \( i + 1 \). If the top bit is set prior to this move it will overflow to create an \( x^8 \) term, in which case the modular polynomial is added to cancel this additional bit, leaving a result that fits within a single byte.

For example, multiplying \( \{11001100\} \) by \( x \) gives an initial result is \( \text{1}\{10010000\} \). The ‘overflow’ bit is then removed by adding \( \text{1}\{00011011\} \), the modular polynomial, using an exclusive-or operation to give a final result of \( \{10010111\} \).

By repeating this process, a finite field element can be multiplied by all powers of \( x \) from 0 to 7. Multiplication of this element by any other field element can then be achieved by
adding the results for the appropriate powers of \( x \). For example, Table 1 carries out this calculation for the product of the field elements \{57\} and \{83\} to give \{c1\}.

<table>
<thead>
<tr>
<th>( p )</th>
<th>( {57} \cdot x^p )</th>
<th>( \oplus m(x) )</th>
<th>( {57} \cdot x^p )</th>
<th>bit p of {83} ( \rightarrow \oplus )</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{01010111}</td>
<td>{01010111}</td>
<td>1 {01010111}</td>
<td>{01010111}</td>
<td>{01010111}</td>
</tr>
<tr>
<td>1</td>
<td>{10101110}</td>
<td>{10101110}</td>
<td>1 {10101110}</td>
<td>{10101110}</td>
<td>{11111001}</td>
</tr>
<tr>
<td>2</td>
<td>{10101111} {00011011}</td>
<td>{01000111}</td>
<td>0 {00000000}</td>
<td>{11111001}</td>
<td>{11111001}</td>
</tr>
<tr>
<td>3</td>
<td>{10001110}</td>
<td>{10001110}</td>
<td>0 {00000000}</td>
<td>{11111001}</td>
<td>{11111001}</td>
</tr>
<tr>
<td>4</td>
<td>{10001110} {00011011}</td>
<td>{00000111}</td>
<td>0 {00000000}</td>
<td>{11111001}</td>
<td>{11111001}</td>
</tr>
<tr>
<td>5</td>
<td>{00001110}</td>
<td>{00001110}</td>
<td>0 {00000000}</td>
<td>{11111001}</td>
<td>{11111001}</td>
</tr>
<tr>
<td>6</td>
<td>{00011100}</td>
<td>{00011100}</td>
<td>0 {00000000}</td>
<td>{11111001}</td>
<td>{11111001}</td>
</tr>
<tr>
<td>7</td>
<td>{00111000}</td>
<td>{00111000}</td>
<td>1 {00111000}</td>
<td>{11000000}</td>
<td>{11000000}</td>
</tr>
</tbody>
</table>

Table 1 – Finite field multiply \{57\} \( \cdot \) \{83\}

2.4 Finite Field Multiplication Using Tables

When certain finite field elements (known as generators) are repeatedly multiplied to produce a list of their powers, \( g^p \), they progressively generate all 255 non-zero elements in the field. When \( p \) reaches 256 the original field element recurs, indicating that \( g^{255} \) is equal to \{01\}. The \( p \) values for each field element can be thought of as logarithms and these provide a way of converting multiplication into addition. Hence the two elements \( a = g^\alpha \) and \( b = g^\beta \) have the product \( a \cdot b = g^{\alpha + \beta} \). With a ‘logarithm’ table listing the power of the generator for each finite field element we can hence find the powers \( \alpha \) and \( \beta \) corresponding to the elements \( a \) and \( b \) and add these values to find the power of \( g \) for the result. A reverse table can then be used to look up the product element.

Since the two initial power values can each be as high as 255, their sum may be greater than 255 but if this occurs, 255 can be subtracted from the value to bring it into the range of the tables because \( g^{255} = \{01\} \). Although decimal exponents have been used in this explanation, all exponents in what follows are in hexadecimal notation.

\[
\begin{array}{cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
For the Rijndael field \( \{03\} \) is a generator that yields Table 2 and Table 3. Using the previous example, Table 2 shows that \( \{57\} = \{03\}^{(62)} \) and \( \{83\} = \{03\}^{(50)} \) (where the brackets on the exponents identify them as hexadecimal numbers). This gives the product as \( \{57\} \cdot \{83\} = \{03\}^{(62)+(50)} \) and since \( (62) + (50) = (b2) \) in hexadecimal, Table 3 gives the result \( \{e1\} \), as before. These tables can also be used to find the inverses of field elements since \( g(x) \) has the inverse \( g^{(ff)-(x)} \). Hence the element \( \{af\} = \{03\}^{b7} \) has the inverse \( g^{(ff)-(b7)} = g^{(48)} = \{62\} \). All elements except \( \{00\} \) have inverses.

### 2.5 Polynomials with Coefficients in GF(256)

Four term polynomials can be defined with coefficients that are finite field elements as:

\[
a(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0
\]

(2.5.1)

where the four coefficients, each represented by a byte, will be denoted as a 32-bit word in the form\([a_3, a_2, a_1, a_0]\). With a second polynomial:

\[
b(x) = b_3 x^3 + b_2 x^2 + b_1 x + b_0
\]

(2.5.2)

addition can be performed by adding the finite field coefficients of like powers of \( x \), which corresponds to an XOR operation between the corresponding bytes in each of the words or an XOR of the complete 32-bit word values (note that the variable \( x \) here is different to that used in the definition of individual finite field elements).

Multiplication is achieved by algebraically expanding the polynomial product and collecting like powers of \( x \) to give:

\[
c(x) = c_6 x^6 + c_5 x^5 + c_4 x^4 + c_3 x^3 + c_2 x^2 + c_1 x + c_0
\]

(2.5.3)

where:

\[
c_0 = a_0 \cdot b_0
\]

\[
c_1 = a_1 \cdot b_0 \oplus a_0 \cdot b_1
\]

\[
c_2 = a_2 \cdot b_0 \oplus a_1 \cdot b_1 \oplus a_0 \cdot b_2
\]

\[
c_3 = a_3 \cdot b_0 \oplus a_2 \cdot b_1 \oplus a_1 \cdot b_2 \oplus a_0 \cdot b_3
\]

\[
c_4 = a_3 \cdot b_1 \oplus a_2 \cdot b_2 \oplus a_1 \cdot b_3
\]

\[
c_5 = a_3 \cdot b_2 \oplus a_2 \cdot b_3
\]

\[
c_6 = a_3 \cdot b_3
\]

(2.5.4)
Where \( \cdot \) and \( \oplus \) denote finite field multiplication and addition (XOR) respectively. This result requires six bytes to represent its coefficients but it can be reduced modulo a degree 4 polynomial to produce a result that is of degree less than 4.

In Rijndael the polynomial used is \( x^4 + 1 \) and reduction produces the following polynomial coefficients:

\[
\begin{align*}
d_3 &= a_3 \cdot b_0 \oplus a_2 \cdot b_1 \oplus a_1 \cdot b_2 \oplus a_0 \cdot b_3 \\
d_2 &= a_2 \cdot b_0 \oplus a_1 \cdot b_1 \oplus a_0 \cdot b_2 \oplus a_3 \cdot b_3 \\
d_1 &= a_1 \cdot b_0 \oplus a_0 \cdot b_1 \oplus a_3 \cdot b_2 \oplus a_2 \cdot b_3 \\
d_0 &= a_0 \cdot b_0 \oplus a_3 \cdot b_1 \oplus a_2 \cdot b_2 \oplus a_1 \cdot b_3
\end{align*}
\] (2.5.5)

If one of the polynomials is fixed, this can conveniently be written in matrix form as:

\[
\begin{bmatrix}
d_3 \\
d_2 \\
d_1 \\
d_0
\end{bmatrix} =
\begin{bmatrix}
a_0 & a_1 & a_2 & a_3 \\
a_3 & a_0 & a_1 & a_2 \\
a_2 & a_3 & a_0 & a_1 \\
a_1 & a_2 & a_3 & a_0
\end{bmatrix}
\begin{bmatrix}
b_3 \\
b_2 \\
b_1 \\
b_0
\end{bmatrix}
\] (2.5.6)

Because \( x^4 + 1 \) is not an irreducible polynomial, not all polynomial multiplications are invertible. For Rijndael, however, a polynomial that has an inverse has been chosen:

\[
\begin{align*}
a(x) &= \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\} \\
a^{-1}(x) &= \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}
\end{align*}
\] (2.5.7)

Another polynomial that Rijndael uses has \( a_0 = a_2 = a_3 = \{00\} \) and \( a_1 = \{01\} \), which is the polynomial \( x \). Inspection of (2.5.6) above will show that its effect is to form the output word by rotating the bytes in the input word so that \([b_3, b_2, b_1, b_0]\) is transformed into \([b_2, b_1, b_0, b_3]\) with bytes moving to higher index positions and the top byte wrapping round to the lowest position. Higher powers of \( x \) correspond to the other cyclic permutations of the four bytes within a 32-bit word. The \texttt{RotWord} function that is used in the key schedule corresponds to \( x^3 \).

3. The Cipher

At the start of the cipher the cipher input is copied into the internal state using the conventions described in Section 1.4. An initial round key is then added and the state is then transformed by iterating a \texttt{round function} in a number of cycles. The number of cycles \( N_n \) varies with the key length and block size. On completion the final state is copied into the cipher output using the same conventions.

The round function is parameterised using a round key which consists of an \( N_c \) word subarray from the key schedule. The latter is considered either as a one-dimensional array of 32-bit words or an array of round keys with a structure and initialisation as described in section 1.5. In general the length of the cipher input, the cipher output and the cipher state, \( N_c \), measured in multiples of 32 bits, is 4, 5, 6, 7 or 8 but the AES standard only allows a length of 4. The length of the cipher key, \( N_k \) as the same values but only lengths of 4, 6 or 8 are allowed in the AES standard.

The cipher is described in the following pseudo code with the individual transformations and the key schedule described subsequently. Here the key schedule is treated as an array of \( N_n + 1 \) individual round keys, each of which is itself an array of \( N_c \) words.
Cipher(byte in[4*Nc], byte out[4*Nc], word k[Nn+1,Nc], Nc, Nn)
Begin
    byte state[4,Nc] // The notation k[Nn+1,Nc] above indicates that
    // the array k contains Nn + 1 individual round
    state = in // keys that are each arrays of Nc words
    XorRoundKey(state, k[0,-], Nc) // k[0,-] = k[0..Nc-1]
    for round = 1 step 1 to Nn – 1
        SubBytes(state, Nc)
        ShiftRows(state, Nc)
        MixColumns(state, Nc)
        XorRoundKey(state, k[round,-], Nc) // k[round*Nc..(round+1)*Nc-1]
    end for
    SubBytes(state, Nc)
    ShiftRows(state, Nc)
    XorRoundKey(state, k[Nn,-], Nc) // k[Nn*Nc..(Nn+1)*Nc-1]
    out = state
end

The number of rounds for the cipher (Nn) varies with the block length and the key length
as shown in the following table. Remember that for AES the block size, Nc, is fixed at 4
and the key, Nk, can only have the lengths 4, 6 or 8.

<table>
<thead>
<tr>
<th>Nn</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 4 – The number of rounds as a function of block and key size

3.1 The SubBytes Transformation

The SubBytes transformation is a non-linear byte substitution that acts on every byte of
the state in isolation to produce a new byte value using an S-box substitution table. The
action of this transformation is illustrated in Figure 2 for a block size of 6.

\[
S_{\text{r,c}} = S_{\text{r,c}}' = \begin{cases} 
S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} & S_{0,4} & S_{0,5} \\
S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} & S_{1,4} & S_{1,5} \\
S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} & S_{2,4} & S_{2,5} \\
S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} & S_{3,4} & S_{3,5} \\
\end{cases}
\]

Figure 2 – SubBytes acts on every byte in the state in isolation

This substitution, which is invertible, is constructed by composing two transformations:

1. First the multiplicative inverse in the finite field described earlier (with element \{00\}
mapped to itself).

2. Second the affine transformation over GF(2) defined by:

\[
b'_i = b_i \oplus b_{(i+4)\mod 8} \oplus b_{(i+5)\mod 8} \oplus b_{(i+6)\mod 8} \oplus b_{(i+7)\mod 8} \oplus c_i \quad (3.1.1)
\]

for 0 ≤ i < 8 where \(b_i\) is bit \(i\) of the byte and \(c_i\) is bit \(i\) of a byte \(c\) with the value
\{63\} or \{01100011\}. Here and elsewhere a prime on a variable on the left of an
equation indicates that its value is to be updated with the value on the right.

In matrix form the latter component of the S-box transformation can be expressed as:
The ShiftRows Transformation

A Specification for The AES Algorithm
Rijndael (by Joan Daemen & Vincent Rijmen)

3.2 The ShiftRows Transformation

The ShiftRows transformation operates individually on each of the last three rows of the state by cyclically shifting the bytes in the row such that:

\[ s'_{r,c} = s_{r, [c + h(r, Nc)] \mod Nc} \text{ for } 0 \leq r < 4 \text{ and } 0 \leq c < Nc \]  

(3.2.1)

where the shift amount \( h(r, Nc) \) depends on row number \( r \) and block length as follows:

<table>
<thead>
<tr>
<th>( h(r, Nc) )</th>
<th>row (r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4, 5, 6</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>7</td>
<td>1, 2, 4</td>
</tr>
<tr>
<td>8</td>
<td>1, 3, 4</td>
</tr>
</tbody>
</table>

Table 6 – Shift offsets for different rows and block lengths

This has the effect of moving bytes to lower positions in the row except that the lowest bytes wrap around into the top of the row (note that a prime on a variable indicates an
updated value). The action of this transformation is illustrated in Figure 3 for a cipher block size of 6.

\[
\begin{array}{cccccccc}
S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} & S_{0,4} & S_{0,5} \\
S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} & S_{1,4} & S_{1,5} \\
S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} & S_{2,4} & S_{2,5} \\
S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} & S_{3,4} & S_{3,5} \\
\end{array}
\]  \rightarrow  \begin{array}{cccccccc}
\text{ShiftRows} & \rightarrow & \begin{array}{cccccccc}
S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} & S_{0,4} & S_{0,5} \\
S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} & S_{1,4} & S_{1,5} \\
S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} & S_{2,4} & S_{2,5} \\
S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} & S_{3,4} & S_{3,5} \\
\end{array}
\end{array}
\]

Figure 3 – ShiftRows acts independently on rows in the state

The pseudo code for this transformation is as follows.

\[
\text{ShiftRows}(\text{byte state}[4,Nc], Nc) \\
\text{begin} \\
\text{byte t}[Nc] \\
\text{for } r = 1 \text{ step 1 to 3} \\
\text{for } c = 0 \text{ step 1 to } Nc - 1 \\
\quad \text{t}[c] = \text{state}[r, (c + h(r,Nc)) \text{ mod } Nc] \\
\text{end for} \\
\text{for } c = 0 \text{ step 1 to } Nc - 1 \\
\quad \text{state}[r,c] = \text{t}[c] \\
\text{end for} \\
\text{end}
\]

3.3 The MixColumns Transformation

The MixColumns transformation acts independently on every column of the state and treats each column as a four-term polynomial as described in Section 2.

In matrix form the transformation used given in equation (3.3.1), where all the values are finite field elements as discussed in Section 2.

\[
\begin{bmatrix}
S_{3,c}' \\
S_{2,c}' \\
S_{1,c}' \\
S_{0,c}'
\end{bmatrix} = 
\begin{bmatrix}
\{02\} & \{01\} & \{01\} & \{03\} \\
\{03\} & \{02\} & \{01\} & \{01\} \\
\{01\} & \{03\} & \{02\} & \{01\} \\
\{01\} & \{01\} & \{03\} & \{02\}
\end{bmatrix}
\begin{bmatrix}
S_{3,c} \\
S_{2,c} \\
S_{1,c} \\
S_{0,c}
\end{bmatrix} \quad \text{for } 0 \leq c < Nc \tag{3.3.1}
\]

The action of this transformation is illustrated in Figure 4 for a cipher block size of 6.

\[
\begin{array}{cccccccc}
S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} & S_{0,4} & S_{0,5} \\
S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} & S_{1,4} & S_{1,5} \\
S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} & S_{2,4} & S_{2,5} \\
S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} & S_{3,4} & S_{3,5} \\
\end{array}
\]  \rightarrow  \begin{array}{cccccccc}
\text{Mix Columns} & \rightarrow & \begin{array}{cccccccc}
S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} & S_{0,4} & S_{0,5} \\
S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} & S_{1,4} & S_{1,5} \\
S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} & S_{2,4} & S_{2,5} \\
S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} & S_{3,4} & S_{3,5} \\
\end{array}
\end{array}
\]

Figure 4 – MixColumns acts independently on each column in the state

The pseudo code for this transformation is as follows, where the function \(\text{FFmul}(x, y)\) returns the product of two finite field elements \(x\) and \(y\).
MixColumns(byte state[4,Nc], Nc)
begin
byte t[4]
for c = 0 step 1 to Nc - 1
for r = 0 step 1 to 3
  t[r] = state[r,c]
end for
for r = 0 step 1 to 3
  state[r,c] = FFmul(0x02, t[r]) xor
               FFmul(0x03, t[(r + 1) mod 4]) xor
               t[(r + 2) mod 4] xor t[(r + 3) mod 4]
end for
end for
end

3.4 The XorRoundKey Transformation

In the XorRoundKey transformation \( Nc \) words from the key schedule (the round key described later) are each added (XOR’d) into the columns of the state so that:

\[
[b_{3c}, b_{2c}, b_{1c}, b_{0c}]' = [b_{3c}, b_{2c}, b_{1c}, b_{0c}] \oplus [k_{round,c}] \quad \text{for } 0 \leq c < Nc
\]

(3.4.1)

where the round key words \( k_{round,c} \) (shortened to \( k_c \) in the diagram below) will be described later. The round number, \( round \), is in the range \( 0 \leq \text{round} < Nn \), with the value of 0 being used to denote the initial round key that is applied before the round function.

The action of this transformation is illustrated in Figure 5 for a cipher block size of 6. The byte address within each word of the key schedule is that described in Section 1.4.

The pseudo code for this transformation is as follows, where \( \text{xbyte}(r, w) \) extracts byte \( r \) from word \( w \).

XorRoundKey(byte state[4,Nc], word k[round,*], Nc)
Begin
for c = 0 step 1 to Nc - 1
  for r = 0 step 1 to 3
    state[r,c] = state[r,c] xor xbyte(r, k[round,c])
  end for
end for
end

4. The Key Schedule

The round keys are derived from the cipher key by means of a key schedule with each round requiring \( Nc \) words of key data which, with an additional initial set, makes a total of \( Nc(Nn + 1) \) words, where \( Nn \) is the number of cipher rounds. This key schedule is considered either as a one dimensional array \( k \) of \( Nc(Nn + 1) \) 32-bit words with an index \( i \) in the range \( 0 \leq i < Nc(Nn + 1) \) or as a two dimensional array \( k[n,c] \) of \( Nn + 1 \) round keys, each of which individually consists of a sub-array of \( Nc \) words.

The expansion of the input key into the key schedule proceeds according to the following pseudo code. The function \( \text{SubWord}(x) \) gives an output word for which the S-box substitution has been individually applied to each of the four bytes of its input \( x \). The
function $\text{RotWord}(x)$ converts an input word $[b_3, b_2, b_1, b_0]$ to an output $[b_0, b_3, b_2, b_1]$. The word array $\text{Rcon}[i]$ contains the values $[0, 0, 0, x^{i-1}]$ with $x^{i-1}$ being the powers of $x$ in the field GF(256) discussed in section 2.3 (note that the index $i$ starts at 1).

$$\text{KeyExpansion}(\text{byte } \text{key}[4*Nk], \text{word } k[Nn+1,Nc], \text{Nc, Nk, Nn})$$

$$\begin{aligned} &\text{begin} \\
&\quad i = 0 \\
&\quad \text{while } (i < Nk) \\
&\quad \quad k[i] = \text{word } [ \text{key}[4*i+3], \text{key}[4*i+2], \text{key}[4*i+1], \text{key}[4*i] ] \\
&\quad \quad i = i + 1 \\
&\quad \text{end while} \\
&\quad i = Nk \\
&\quad \text{while } (i < \text{Nc} \times (\text{Nn + 1})) \\
&\quad \quad \text{word } \text{temp} = k[i - 1] \\
&\quad \quad \text{if } (i \text{ mod Nk} = 0) \text{ or } ((\text{Nk} > 6) \text{ and } (i \text{ mod Nk} = 4)) \\
&\quad \quad \quad \text{temp} = \text{SubWord}(\text{temp}) \\
&\quad \quad \text{end if} \\
&\quad \quad \text{if } (i \text{ mod Nk} = 0) \\
&\quad \quad \quad \text{temp} = \text{RotWord}(\text{temp}) \text{ xor } \text{Rcon}[i / \text{Nk}] \\
&\quad \quad \text{end if} \\
&\quad k[i] = k[i - \text{Nk}] \text{ xor } \text{temp} \\
&\quad i = i + 1 \\
&\quad \text{end while} \\
&\text{end} \\
\end{aligned}$$

Note that this key schedule, which is illustrated in Figure 6 for $Nk = 4$ and $Nc = 6$, can be generated ‘on-the fly’ if necessary using a buffer of $\text{max}(Nc, Nk)$ words. It can also be split into separate, somewhat simpler, key schedules for $Nk \leq 6$ and $Nk > 6$ respectively.

<table>
<thead>
<tr>
<th>Key Schedule</th>
<th>Round Key 0</th>
<th>Round Key 1</th>
<th>Round Key 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_0$</td>
<td>$k[0,*]$</td>
<td>$k[1,*]$</td>
<td>$k[2,*]$</td>
</tr>
<tr>
<td>$k_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k_2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k_3$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k_4$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k_5$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k_6$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k_7$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k_8$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k_9$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k_{10}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k_{11}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k_{12}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k_{13}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k_{14}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k_{15}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k_{16}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k_{17}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6 – The key schedule and round key selection for $Nk = 4$ and $Nc = 6$

### 5. The Inverse Cipher

The inversion of the cipher code presented in section 3 is straightforward and provides the following pseudo code for the inverse cipher.

$$\text{InvCipher}(\text{byte } \text{in}[4*Nc], \text{byte } \text{out}[4*Nc], \text{word } k[Nn+1,Nc], \text{Nc, Nn})$$

$$\begin{aligned} &\text{begin} \\
&\quad \text{byte } \text{state}[4,Nc] \\
&\quad \text{state} = \text{in} \\
&\quad \text{XorRoundKey}(\text{state}, k[Nn,-], \text{Nc}) \quad \text{// } k[Nn*Nc..(Nn+1)*Nc-1] \\
&\quad \text{for } \text{round} = \text{Nn} - 1 \text{ step } -1 \text{ to } 1 \\
&\quad \quad \text{InvShiftRows}(\text{state}, \text{Nc}) \\
&\quad \quad \text{InvSubBytes}(\text{state}, \text{Nc}) \\
&\quad \quad \text{XorRoundKey}(\text{state}, k[\text{round},-], \text{Nc}) \quad \text{// } k[\text{round}*Nc..(\text{round}+1)*\text{Nc}-1] \\
&\quad \quad \text{InvMixColumns}(\text{state}, \text{Nc}) \\
&\quad \text{end for} \\
&\quad \text{InvShiftRows}(\text{state}, \text{Nc}) \\
&\quad \text{InvSubBytes}(\text{state}, \text{Nc}) \\
&\quad \text{XorRoundKey}(\text{state}, k[0,-], \text{Nc}) \quad \text{// } k[0..\text{Nc}-1] \\
&\quad \text{out} = \text{state} \\
&\text{end} \\
\end{aligned}$$
5.1 The Inverse ShiftRows Transformation

The \texttt{InvShiftRows} transformation operates individually on each of the last three rows of the state cyclically shifting the bytes in the row such that:

$$s'_{r,(c+h(r,Nc)) \mod Nc} = s_{r,c} \quad \text{for} \quad 0 \leq r < 4 \quad \text{and} \quad 0 \leq c < Nc$$ \hfill (5.1.1)

where the cyclic shift values \(h(r,Nc)\) are given in Table 6. The pseudo code for this transformation is as follows.

\[
\text{InvShiftRows}(\text{byte state}[4,Nc], Nc) \\
\begin{align*}
\text{begin} \\
\text{byte } t[Nc] \\
\text{for } r = 1 \text{ step 1 to 3} \\
\text{for } c = 0 \text{ step 1 to } Nc - 1 \\
\text{t[\{c + h(r,Nc)\} \mod Nc] = state[r,c]} \\
\text{end for} \\
\text{for } c = 0 \text{ step 1 to } Nc - 1 \\
\text{state[r,c] = t[c]} \\
\text{end for} \\
\text{end for} \\
\text{end}
\]

5.2 The Inverse SubBytes Transformation

The inverse S-box table needed for the inverse \texttt{InvSubBytes} transformation is given in Section 3.1. The pseudo code for this transformation is as follows:

\[
\text{InvSubBytes}(\text{byte state}[4,Nc], Nc) \\
\begin{align*}
\text{begin} \\
\text{for } r = 0 \text{ step 1 to 3} \\
\text{for } c = 0 \text{ step 1 to } Nc - 1 \\
\text{state[r,c] = InvSbox[state[r,c]]} \\
\text{end for} \\
\text{end for} \\
\text{end}
\]

Table 7 gives the full inverse S-box, the inverse of the affine transformation (3.1.1) being:

$$b'_i = b_{(i+2) \mod 8} \oplus b_{(i+5) \mod 8} \oplus b_{(i+7) \mod 8} \oplus d_1 \quad \text{where byte } d = \{05\} \hfill (5.2.1)$$

\[
\begin{array}{cccccccccccccccccccccccccc}
\text{hex} & \text{x} & \text{invSbox} & \text{y} & \text{hex} & \text{y} \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & a & b & c & d & e & f \\
0 & 52 & 09 & 6a & d5 & 30 & 36 & a5 & 38 & bf & 40 & a3 & 9e & 81 & f3 & d7 & fb \\
1 & 7c & e3 & 39 & 82 & 9b & 2f & ff & 87 & 34 & 8e & 43 & 44 & c4 & de & e9 & cb \\
2 & 54 & 7b & 94 & 32 & a6 & c2 & 23 & 3e & 4c & 95 & 0b & 42 & fa & c3 & 4e \\
3 & 08 & 2e & a1 & 66 & 28 & d9 & 24 & b2 & 76 & 5b & a2 & 49 & 6d & 8b & d1 & 25 \\
4 & 72 & f8 & 6f & 64 & 86 & 68 & 98 & 16 & d4 & a4 & 5c & cc & 5d & 6b & 92 \\
5 & 6c & 70 & 48 & 50 & fd & ed & b9 & da & 5e & 15 & 46 & 57 & a7 & 8d & 9d & 84 \\
6 & 90 & d8 & ab & 00 & 8c & bc & d3 & 0a & f7 & e4 & 58 & 05 & b8 & b3 & 45 & 06 \\
7 & d0 & 2c & 1e & 8f & ca & 3f & 0f & 02 & c1 & af & bd & 03 & 01 & 13 & 8a & 6b \\
8 & 3a & 91 & 11 & 41 & 4f & 67 & dc & ea & 97 & f2 & cf & ce & f0 & b4 & e6 & 73 \\
9 & 96 & ac & 74 & 22 & e7 & ad & 35 & 85 & e2 & f9 & 37 & e8 & 1c & 75 & df & 6e \\
a & 47 & f1 & 1a & 71 & 1d & 29 & c5 & 89 & 6f & b7 & 62 & 0e & aa & 18 & be & 1b \\
b & fc & 56 & 3e & 4b & c6 & d2 & 79 & 20 & 9a & db & c0 & fe & 78 & cd & 5a & f4 \\
c & 1f & dd & a8 & 33 & 88 & 07 & c7 & 31 & b1 & 12 & 10 & 59 & 27 & 80 & ec & 5f \\
d & 60 & 51 & 7f & a9 & 19 & b5 & 4a & 0d & 2d & e5 & 7a & 9f & 93 & c9 & 9c & ef \\
e & a0 & e0 & 3b & 4d & ae & 2a & f5 & b0 & c8 & eb & bb & 3c & 83 & 53 & 99 & 61 \\
f & 17 & 2b & 04 & 7e & ba & 77 & d6 & 26 & e1 & 69 & 14 & 63 & 55 & 21 & 0c & 7d \\
\end{array}
\]

Table 7 – The Inverse Substitution Table – InvSbox[xy] (in hexadecimal)

5.3 The Inverse XorRoundKey Transformation

The \texttt{XorRoundKey} transformation is its own inverse.
5.4 The Inverse MixColumns Transformation

The InvMixColumns transformation acts independently on every column of the state and treats each column as a four-term polynomial as described in Section 2.6. In matrix form the transformation used given in equation (5.4.1), where all the values are finite field elements as discussed in Section 2.

The pseudo code for this transformation is as follows, where the function FFmul(x, y) returns the product of two finite field elements x and y.

InvMixColumns(byte block[4,Nc], Nc)
begin
byte t[4]
for c = 0 step 1 to Nc – 1
  for r = 0 step 1 to 3
    t[r] = block[r,c]
  end for
  for r = 0 step 1 to 3
    block[r,c] =
      FFmul(0x0e, t[r]) xor
      FFmul(0x0b, t[(r + 1) mod 4]) xor
      FFmul(0x0d, t[(r + 2) mod 4]) xor
      FFmul(0x09, t[(r + 3) mod 4])
  end for
end for
end

5.5 The Equivalent Inverse Cipher

The inverse cipher uses the same key schedule as the forward cipher (in reverse) but its form is different. However a series of transformations can be applied to transform the inverse cipher to match the form of the forward cipher. This is possible because the order of some operations in the inverse cipher can be changed without changing the final result.

For example the order of the SubBytes and ShiftRows transformations does not matter because SubBytes changes the value of bytes without changing their positions whereas ShiftRows does the exact opposite. Moreover, the order of the XorRoundKey and InvMixColumns operations can be inverted to put the forward and inverse ciphers in the same form provided that an adjustment is made to the key schedule. The order of round key addition and column mixing can be changed because the column mixing operation is linear with respect to the column input so that:

\[ \text{InvMixColumns(state } \oplus \text{k)} = \text{InvMixColumns(state)} \oplus \text{InvMixColumns(k)} \]

where k represents a round key in the form of a state array. Hence, provided that an inverse column mixing operation is performed on appropriate words (columns) of the decryption key schedule, the order of these transformations can be reversed during decryption. Note, however, that this operation is not performed on the first and last round keys (the first and last Nc words of the key schedule) since these do not operate in association with the column-mixing step.

The importance of this transformation is that the structure of the forward cipher allows the round function to be expressed in an efficient form for implementation. By transforming the inverse cipher into the same sequence of operations as the cipher itself, it can be implemented in the same way, thereby achieving this efficiency.
In this modified form the inverse cipher is as follows (with the modified decryption key schedule in the word array \(dk[Nn+1,Nc]\)).

\[
\text{EqInvCipher}(\text{byte in}[4*Nc], \text{byte out}[4*Nc], \text{word } dk[Nn+1,Nc], Nc, Nn) \\
\begin{align*}
\text{begin} \\
\text{byte state}[4,Nc] \\
&= \text{in} \\
&\text{XorRoundKey(state, } dk[Nn,-], Nc) \quad // \ dk[Nn*Nc..(Nn+1)*Nc-1] \\
&\text{for } round = Nn - 1 \text{ step -1 to 1} \\
&\quad \text{InvSubBytes(state, } Nc) \\
&\quad \text{InvShiftRows(state, } Nc) \\
&\quad \text{InvMixColumns(state, } Nc) \\
&\quad \text{XorRoundKey(state, } dk[round,-], Nc) \quad // \ dk[round*Nc..(round+1)*Nc-1] \\
&\text{end for} \\
&\text{InvSubBytes(state, } Nc) \\
&\text{InvShiftRows(state, } Nc) \\
&\text{XorRoundKey(state, } dk[0,-], Nc) \quad // \ dk[0..Nc-1] \\
&\text{out} = \text{state} \\
\text{end}
\end{align*}
\]

where the following pseudo code is added to the end of the key expansion step (this can be made more efficient if encryption and decryption are not required simultaneously).

\[
\begin{align*}
\text{for } round = 0 \text{ step 1 to } Nn \\
&\quad dk[i,*] = k[i,*] \quad // \ copy Nc \text{ words at a time} \\
\text{end for} \\
\text{for } round = 1 \text{ step 1 to } Nn - 1 \\
&\quad \text{InvMixColumns(dk[round,*])} \quad // \ note \ implicit \ change \ of \ type \\
\text{end for}
\end{align*}
\]

Note that, since \text{InvMixColumns} operates on a two-dimensional array of bytes while the round keys are held in an array of words, the call to \text{InvMixColumns} in this pseudo code sequence involves a change of type. This requires care with byte order conventions.

6. Implementation Issues

6.1 Implicit Assumptions

While hardware implementations of Rijndael can treat the input, output and cipher key inputs as bit sequences, software implementations will almost always to treat these entities as arrays of 8-bit bytes. Equally, while a hardware implementation will have to include a description of how Rijndael inputs and outputs are interfaced, a software implementation will often operate in an environment where Rijndael’s two key enumerations – the enumeration of bits within 8-bit bytes and the enumeration of bytes within arrays – are already defined.

Where the environment in which Rijndael is implemented provides both for 8-bit bytes as addressable entities and for the enumeration of bits within bytes, it is reasonable to assume that Rijndael inputs and outputs will comply with these conventions.

In consequence Rijndael implementations in software should either indicate that this assumption is correct or alternatively undertake one of the following:

(a) convert inputs and outputs to (or from) these standard formats to those being used internally;

(b) document the interface to ensure that users of the implementation know that the inputs and outputs are in non-standard formats.
6.2 Bit Enumerations

In processing bytes to undertake finite field multiplication it is useful to define a function to multiply by \( x \), an operation that involves shifting the value of a byte by one and then performing a conditional XOR operation. If by convention bit 0 is the ‘lowest’ bit in a byte (i.e. it represents a numeric value of 1) then multiplying by \( x \) will correspond to a left shift. This is the most likely situation but it is not unknown for bit 0 to be designated as the ‘highest’ bit in a byte, the bit that represents a numeric value of 128 in decimal, in which case multiplication by \( x \) will correspond to a right shift. When this applies, all byte values will also have their bits reversed so that \{01100011\} or \{63\}, which in former convention would be associated with a numeric value of 0x63 in hexadecimal, will instead be associated with a numeric value of 0xc6. For this reason the terms ‘left’ and ‘right’ when referring to shifts have been avoided in this specification by using the terms ‘up’ and ‘down’ to refer to operations in which bytes at an index position move to higher or lower index positions respectively.

6.3 Bytes within Words

A number of Rijndael operations involve the manipulation of the four 8-bit bytes within a 32-bit word, one such operation being the cyclic shift (rotation) of these four bytes into new positions. Whether the operation of moving bytes to higher array index positions corresponds to a cyclic left or a cyclic right shift for a 32-bit word will depend on how the bytes are organised within words.

On some (‘little-endian’) processors bytes are numbered upwards from the ‘low’ end of 32-bits words and this means that a cyclic shift of bytes to higher array index positions will correspond to a cyclic left shift. But on other (‘big-endian’) processors bytes are numbered upwards starting at the ‘high’ end of a word so that a cyclic shift to higher index positions corresponds to a cyclic right shift.

In consequence care is needed in implementing Rijndael to ensure that the right directions of shifts and rotates are employed for the processor or processors for which an implementation is being designed.

In general these issues can be tackled either by the conversion of input and output values before use or by ensuring that the conventions employed for implementation are those of the architecture on which the cipher will operate.

7. Implementation Techniques

In the pseudo code in this section the following symbols will be used:

- \& bits in result are the AND of the corresponding bits in the two operands
- | bits in result are the OR of the corresponding bits in the two operands
- ^ bits in result are the XOR of the corresponding bits in the two operands
- >> right shift of left operand by amount given by right operand
- << left shift of left operand by amount given by right operand
- <> not equal
- 0x.. hexadecimal value

7.1 Finite Field Multiplication

The basic technique for finite field multiplication is explained in Section 2.4 and is implemented as follows:
byte FFmul(const byte a, const byte b)
begin
  byte aa = a, bb = b, r = 0, t
  while (aa <> 0)
    if ((aa & 1) <> 0)
      r = r ^ bb
    endif
    t = bb & 0x80
    bb = bb << 1
    if (t <> 0)
      bb = bb ^ 0x1b  // the top bit of field polynomial (0x11b) is not needed here since bb is an 8 bit value
    endif
    aa = aa >> 1
  endwhile
  return r
end

But this approach can be quite slow compared with table lookup using the techniques described in Section 2.5. With a 256-byte arrays from tables 2 and 3 we obtain:

byte FFlog[256]  // array from table 2
byte FFpow[256] // array from table 3

byte FFmul(const byte a, const byte b)
begin
  if ((a <> 0) and (b <> 0))
    word t = FFlog[a] + FFlog[b]
    if(t >= 255)
      t = t – 255
    endif
    return FFpow[t]
  else
    return 0
  endif
end

This can be speeded up by doubling the length of the FFpow[] array and setting the values for elements 255 to 509 to the same values as elements 0 to 254 respectively so that FFmul() can be coded as:

byte FFmul(const byte a, const byte b)
begin
  if ((a <> 0) and (b <> 0))
    return FFpow[FFlog[a] + FFlog[b]]
  else
    return 0
  endif
end

In practice many compilers will allow these functions to be specified as inline code and this makes finite field multiplication very efficient.

7.2 Column Mixing

Provided that the state array is arranged appropriately in memory, each of the columns will be a single 32-bit word. If the bytes in such a word are c[3] to c[0] then the mixing operation is:

\[
\begin{align*}
c[3] &= \{02\} \cdot c[3] \oplus \{03\} \cdot c[0] \oplus c[1] \oplus c[2] \\
c[2] &= \{02\} \cdot c[2] \oplus \{03\} \cdot c[3] \oplus c[0] \oplus c[1] \\
c[1] &= \{02\} \cdot c[1] \oplus \{03\} \cdot c[2] \oplus c[3] \oplus c[0] \\
c[0] &= \{02\} \cdot c[0] \oplus \{03\} \cdot c[1] \oplus c[2] \oplus c[3]
\end{align*}
\]

(7.2.1)

where the bytes are updated with the values on the left at the end of this sequence. But since \{03\} \cdot c[0] = \{02\} \cdot c[0] \oplus c[0], this can also be written as:
\[\begin{align*}
    c[3]' &= v \oplus c[3] \oplus \{02\} \cdot (c[3] \oplus c[0]) \\
    c[2]' &= v \oplus c[2] \oplus \{02\} \cdot (c[2] \oplus c[3]) \\
    c[1]' &= v \oplus c[1] \oplus \{02\} \cdot (c[1] \oplus c[2]) \\
    c[0]' &= v \oplus c[0] \oplus \{02\} \cdot (c[0] \oplus c[1])
\end{align*}\] (7.2.2)

where \(v = c[3] \oplus c[2] \oplus c[1] \oplus c[0]\). When the need for temporary storage is taken into account, this code sequence becomes (with temporary variables \(t\), \(u\) and \(v\)):

\[
\begin{align*}
u &= c[1] \land c[0] \\
v &= t \land u \\
c[3] &= c[3] \land v \land FFmul(0x02, c[0] \land c[3]) \\
c[2] &= c[2] \land v \land FFmul(0x02, t) \\
c[1] &= c[1] \land v \land FFmul(0x02, c[2] \land c[1]) \\
c[0] &= c[0] \land v \land FFmul(u)
\end{align*}
\]

Moreover, multiplication by the element \(\{02\}\) is just a shift followed by a conditional exclusive-or operation.

Although this formulation is quite efficient on 8-bit processors, the operations can be speeded up considerably on processors with 32 bit words provided that there are operations that can cyclically rotate the bytes within such words. The functions required are as follows:

- \(\text{rot1}(w)\) moves the bytes in positions 0, 1 and 2 in the word \(w\) to positions 1, 2 and 3 respectively and moves the byte in position 3 to position 0.
- \(\text{rot2}(w)\) moves the bytes in positions 0, 1, 2 and 3 in \(w\) to positions 2, 3, 0 and 1 respectively (or exchanges byte 0 with byte 2 and byte 1 with byte 3).
- \(\text{rot3}(w)\) moves the bytes in positions 1, 2 and 3 in \(w\) to positions 0, 1 and 2 respectively and moves the byte in position 0 to position 3.

Using these operations on each word \(w\) of the state allows the above code sequence on individual bytes to be rewritten as one operation on each word (column) as a whole:

\[
w = \text{rot3}(w) \land \text{rot2}(w) \land \text{rot1}(w) \land \text{FFmulX}(w \land \text{rot3}(w))
\]

where the function \(\text{FFmulX}(w)\) performs a finite field multiplication of each of the four bytes in the word \(w\) by \(\{02\}\). This can be coded to operate in parallel on the four bytes in the word as follows:

\[
\text{word FFmulX(const word w)}
\begin{align*}
\text{begin} \\
\text{word t = w \& 0x80808080} \\
\text{return ((w \land t) \ll 1) \land ((t >> 3) \land (t >> 4) \land (t >> 6) \land (t >> 7))}
\end{align*}
\]

Here the word \(t\) extracts the highest bits from each byte within \(w\), while the term \(w \land t\) extracts the lower 7 bits. The four individual bytes within the latter can then be multiplied by \(\{02\}\) in parallel using a single 32-bit left shift without creating overflows from one byte to the next. The \((t >> 3) \land (t >> 4) \land (t >> 6) \land (t >> 7))\) construction leaves zero bytes within \(t\) unchanged but changes the bytes whose top bits are set to 0x1b. There are several alternative ways of performing this step including, for example \(((u - (u >> 7)) \land 0x1b1b1b1b)\) or \(((u >> 7) * 0x0000001b)\), the most efficient depending on the characteristics of the processor instruction set available for its implementation. Finally, when this value is XOR’ed into the result the effect is that required – namely, the modular polynomial is added to all bytes in which the top bits were originally set.
7.3 Inverse Column Mixing

As with forward column mixing, the inverse operation can be expressed as operations on the four bytes within a column contained within a single 32-bit word.

Provided that the state array is arranged appropriately in memory, each of the columns will be a single 32-bit word. If the bytes in such a word are \(c[3]\) to \(c[0]\) then the mixing operation is:

\[
\begin{align*}
\vec{c}[3] &= \{0e\} \cdot c[3] \oplus \{0b\} \cdot c[0] \oplus \{0d\} \cdot c[1] \oplus \{09\} \cdot c[2] \\
\vec{c}[2] &= \{0e\} \cdot c[2] \oplus \{0b\} \cdot c[3] \oplus \{0d\} \cdot c[0] \oplus \{09\} \cdot c[1] \\
\vec{c}[1] &= \{0e\} \cdot c[1] \oplus \{0b\} \cdot c[2] \oplus \{0d\} \cdot c[3] \oplus \{09\} \cdot c[0] \\
\vec{c}[0] &= \{0e\} \cdot c[0] \oplus \{0b\} \cdot c[1] \oplus \{0d\} \cdot c[2] \oplus \{09\} \cdot c[3]
\end{align*}
\]  

(7.3.1)

At first sight this looks very different to the forward transformation matrix but if we look at the first row, we can rewrite this as:

\[
\begin{align*}
\vec{c}[3] &= \{02\} \oplus \{04\} \oplus \{08\} \cdot \{03\} \oplus \{08\} \cdot \{c[0]\} \\
&\quad \oplus \{01\} \oplus \{04\} \oplus \{08\} \cdot \{c[1]\} \oplus \{01\} \oplus \{08\} \cdot \{c[2]\} \\
&= \{08\} \cdot (c[3] \oplus c[2] \oplus c[1] \oplus c[0]) \oplus \{04\} \cdot (c[3] \oplus c[1]) \\
&\quad \oplus \{(02) \cdot c[3] \oplus \{03\} \cdot c[0] \oplus \{c[1]\} \oplus \{c[2]\}\}
\end{align*}
\]  

(7.3.2)

where \(v = (c[3] \oplus c[2] \oplus c[1] \oplus c[0])\). The inverse transformation is hence:

\[
\begin{align*}
\vec{c}[3] &= \{04\} \cdot (c[3] \oplus c[1]) \oplus \{v \oplus c[3] \oplus \{02\} \cdot \{c[3] \oplus c[0]\}) \\
\vec{c}[2] &= \{04\} \cdot (c[2] \oplus c[0]) \oplus \{v \oplus c[2] \oplus \{02\} \cdot \{c[2] \oplus c[3]\}) \\
\vec{c}[1] &= \{04\} \cdot (c[1] \oplus c[3]) \oplus \{v \oplus c[1] \oplus \{02\} \cdot \{c[1] \oplus c[2]\}) \\
\vec{c}[0] &= \{04\} \cdot (c[0] \oplus c[2]) \oplus \{v \oplus c[0] \oplus \{02\} \cdot \{c[0] \oplus c[1]\})
\end{align*}
\]  

(7.3.3)

which is now similar in form to the forward calculation. The code sequence to implement this with temporary variables \(t, u, v\) and \(w\) is then:

\[
\begin{align*}
t &= \vec{c}[3] \wedge \vec{c}[2] \\
u &= \vec{c}[1] \wedge \vec{c}[0] \\
v &= t \wedge u \\
w &= v \wedge \text{FFmul}(0x08, v) \\
w &= v \wedge \text{FFmul}(0x04, \{c[2]\} \wedge \{c[0]\}) \\
v &= v \wedge \text{FFmul}(0x04, \{c[3]\} \wedge \{c[1]\}) \\
c[3] &= \vec{c}[3] \wedge v \wedge \text{FFmul}(0x02, c[0] \wedge c[3]) \\
c[2] &= \vec{c}[2] \wedge w \wedge \text{FFmul}(0x02, t) \\
c[1] &= \vec{c}[1] \wedge v \wedge \text{FFmul}(0x02, \{c[2]\} \wedge \{c[1]\}) \\
c[0] &= \vec{c}[0] \wedge w \wedge \text{FFmul}(0x02, u)
\end{align*}
\]

As for forward mixing, this calculation can be optimised in situations where 32-bit operations that include rotate instructions are available.

7.4 Implementation Using Tables

Rijndael can be implemented very efficiently on processors with 32-bit words by transforming it in the following way.

Considering a single column (word) of the state and applying the \texttt{SubBytes}, \texttt{ShiftRows}, \texttt{MixColumns} and \texttt{XorRoundKey} transformations in turn gives:

After \texttt{SubBytes}:  

\[
\begin{align*}
&
\begin{array}{c}
\text{Dr. Brian Gladman, v3.16, 1st August 2007}
\end{array}
\end{align*}
\]
\[
\begin{bmatrix}
S_{3,c}' \\
S_{2,c} \\
S_{1,c} \\
S_{0,c}
\end{bmatrix}
= \begin{bmatrix}
S[S_{3,c}] \\
S[S_{2,c}] \\
S[S_{1,c}] \\
S[S_{0,c}]
\end{bmatrix}
\] (7.4.1)

After \textit{ShiftRows}:

\[
\begin{bmatrix}
S_{3,c}'' \\
S_{2,c} \\
S_{1,c} \\
S_{0,c}
\end{bmatrix}
= \begin{bmatrix}
S[S_{3,c}] \cdot (0+2) & S[S_{2,c}] \cdot (01) & S[S_{1,c}] \cdot (01) \\
S[S_{3,c}] \cdot (03) & S[S_{2,c}] \cdot (02) & S[S_{1,c}] \cdot (01) \\
S[S_{3,c}] \cdot (01) & S[S_{2,c}] \cdot (03) & S[S_{1,c}] \cdot (02) \\
S[S_{3,c}] \cdot (01) & S[S_{2,c}] \cdot (03) & S[S_{1,c}] \cdot (02)
\end{bmatrix}
\] (7.4.2)

After \textit{MixColumns}:

\[
\begin{bmatrix}
S_{3,c}'''' \\
S_{2,c} \\
S_{1,c} \\
S_{0,c}
\end{bmatrix}
= \begin{bmatrix}
(02) & (01) & (01) & (03) \\
(03) & (02) & (01) & (01) \\
(01) & (03) & (02) & (01) \\
(01) & (01) & (03) & (02)
\end{bmatrix}
\begin{bmatrix}
S[S_{3,c}] \\
S[S_{2,c}] \\
S[S_{1,c}] \\
S[S_{0,c}]
\end{bmatrix}
\oplus \begin{bmatrix}
k_{3,c} \\
k_{2,c} \\
k_{1,c} \\
k_{0,c}
\end{bmatrix}
\] (7.4.3)

After \textit{XorRoundKey}:

\[
\begin{bmatrix}
S_{3,c}''' \\
S_{2,c} \\
S_{1,c} \\
S_{0,c}
\end{bmatrix}
= \begin{bmatrix}
(02) & (01) & (01) & (03) \\
(03) & (02) & (01) & (01) \\
(01) & (03) & (02) & (01) \\
(01) & (01) & (03) & (02)
\end{bmatrix}
\begin{bmatrix}
S[S_{3,c}] \\
S[S_{2,c}] \\
S[S_{1,c}] \\
S[S_{0,c}]
\end{bmatrix}
\oplus \begin{bmatrix}
k_{3,c} \\
k_{2,c} \\
k_{1,c} \\
k_{0,c}
\end{bmatrix}
\] (7.4.4)

where the shorthand notation \(c(r) = [c + h(r, Nc)] \mod Nc\) with \(c(0) = c\), has been used in the column index \(c\).

Treating this as one complex transformation (i.e. with a single prime), it can be written in column vector form as:

\[
\begin{bmatrix}
S_{3,c}' \\
S_{2,c} \\
S_{1,c} \\
S_{0,c}
\end{bmatrix}
= \begin{bmatrix}
S[S_{3,c}] \\
S[S_{2,c}] \\
S[S_{1,c}] \\
S[S_{0,c}]
\end{bmatrix}
\oplus \begin{bmatrix}
k_{3,c} \\
k_{2,c} \\
k_{1,c} \\
k_{0,c}
\end{bmatrix}
\] (7.4.5)

And if four tables each of 256 32-bit words are defined (for \(0 \leq x < 256\)) as follows:

\[
T_3[x] = \begin{bmatrix}
(02) \cdot S[x] \\
(03) \cdot S[x]
\end{bmatrix}
\quad T_2[x] = \begin{bmatrix}
S[x] \\
S[x]
\end{bmatrix}
\quad T_1[x] = \begin{bmatrix}
(02) \cdot S[x] \\
(03) \cdot S[x]
\end{bmatrix}
\quad T_0[x] = \begin{bmatrix}
(03) \cdot S[x] \\
(02) \cdot S[x]
\end{bmatrix}
\] (7.4.6)

equation (7.4.5) can then be expressed in the form:

\[
\begin{bmatrix}
S_{3,c}' \\
S_{2,c} \\
S_{1,c} \\
S_{0,c}
\end{bmatrix}
= T_3[S_{3,c}(3)] \oplus T_2[S_{2,c}(2)] \oplus T_1[S_{1,c}(1)] \oplus T_0[S_{0,c}(0)] \oplus k_{\text{round,c}}
\] (7.4.7)

where \(c(r) = [c + h(r, Nc)] \mod Nc\), \(c(0) = c\) and \(k_{\text{round,c}}\) is word \(c\) of round key round.

This shows that each column in the output state can be computed using four XOR instructions involving a word from the key schedule and four words from tables that are indexed using four bytes from the input state.
Equation (7.4.7) applies to all but the last round because the latter is different in that the MixColumns step is not present. This means that different tables are required for the last round as follows:

\[
\begin{bmatrix}
S_{3,c} \\
S_{2,c} \\
S_{1,c} \\
S_{0,c}
\end{bmatrix} \quad U_3[x] = \begin{bmatrix} 0 \\ 0 \\ 0 \\ S[x] \end{bmatrix} \\
U_2[x] = \begin{bmatrix} 0 \\ 0 \\ S[x] \\ 0 \end{bmatrix} \\
U_1[x] = \begin{bmatrix} 0 \\ S[x] \\ 0 \\ 0 \end{bmatrix} \\
U_0[x] = \begin{bmatrix} S[x] \\ 0 \\ 0 \\ 0 \end{bmatrix}
\] (7.4.8)

These tables can be implemented directly or can be computed from the S-Box table or by masking the appropriate tables for normal rounds. 4096 bytes (4 x 256 x 4) of table space is needed for the main rounds and this doubles if last round tables are also used. However, the four tables are closely related to each other since \(T_i[x] = \text{rot1}(T_{i-1}[x])\), so the space needed can be reduced by a factor of four at the expense of three additional rotations in the calculation of each column of the state.

The implementation approach described in this section can also be used for the equivalent inverse cipher since this has the same high level structure as the forward cipher. But a different set of tables is needed because the inverse S-Boxes and the inverse column mixing operation have to be used in this case. The byte indexing for the table values is also different for the inverse cipher, namely, \(c(r) = [c - h(r, Nc)] \mod Nc\). For the inverse cipher, the normal round tables are hence:

\[
V_3[x] = \begin{bmatrix} 0e \cdot I[x] \\ 0b \cdot I[x] \\ 0d \cdot I[x] \\ 09 \cdot I[x] \end{bmatrix} \\
V_2[x] = \begin{bmatrix} 09 \cdot I[x] \\ 0e \cdot I[x] \\ 0b \cdot I[x] \\ 0d \cdot I[x] \end{bmatrix} \\
V_1[x] = \begin{bmatrix} 0d \cdot I[x] \\ 09 \cdot I[x] \\ 0e \cdot I[x] \\ 0b \cdot I[x] \end{bmatrix} \\
V_0[x] = \begin{bmatrix} 0b \cdot I[x] \\ 0d \cdot I[x] \\ 09 \cdot I[x] \\ 0e \cdot I[x] \end{bmatrix}
\] (7.4.9)

with \(I[x] = S^{-1}[x]\), which allows the equivalent inverse cipher round transformation to be expressed as:

\[
\begin{bmatrix}
S_{3,c} \\
S_{2,c} \\
S_{1,c} \\
S_{0,c}
\end{bmatrix}' = V_3[S_{3,c(3)}] \oplus V_2[S_{2,c(2)}] \oplus V_1[S_{1,c(1)}] \oplus V_0[S_{0,c(0)}] \oplus k_{\text{round},c}
\] (7.4.10)

where \(c(r) = [c - h(r, Nc)] \mod Nc\), \(c(0) = c\) and \(k_{\text{round},c}\) is word \(c\) of round key \(\text{round}\) for the equivalent inverse cipher. The inverse last round tables \((W)\) match equation (7.4.8) with \(I[x] = S^{-1}[x]\) replacing \(S[x]\).

8. Acknowledgements

This specification was originally written as an input to the AES FIPS development process but it has been developed further since then. I would like to acknowledge the contributions of Joan Daemen, Vincent Rijmen, Jim Foti, Elaine Barker, Morris Dworkin, Lawrence Bassham, Paulo Barreto, Bryan Olson, David Hopwood and Doug Gwynn.

9. References


10. Errors

This specification has been produced from the base document referenced in section 9. It has no formal status but the author would be grateful for reports of any errors in it to brg@gladman.plus.com. C implementations of Rijndael by the author are available at:

http://fp.gladman.plus.com/cryptography_technology/index.htm
11. An Example of Cipher Operation

The diagram shows the hexadecimal values in the state array as the cipher progresses for a cipher input length ($N_c$) of 4 and a cipher key length ($N_k$) of 4. The notation for the following inputs is given at the start of Section 12.

<table>
<thead>
<tr>
<th>Input</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3243f6a8885a308d313198ae0370734$</td>
<td>$2b7e151628aed2a6abf7158889cf4f3c$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>round number</th>
<th>start of round</th>
<th>after subbytes</th>
<th>after shiftrows</th>
<th>after mixcolumns</th>
<th>round key value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$19a09ae9$</td>
<td>$d4e0b81e$</td>
<td>$d4e0b81e$</td>
<td>$d0e04828$</td>
<td>$2b28ab09$</td>
</tr>
<tr>
<td>2</td>
<td>$a82d34$</td>
<td>$e23d5328$</td>
<td>$d21358$</td>
<td>$e5a97a4f$</td>
<td>$16a838c$</td>
</tr>
<tr>
<td>3</td>
<td>$a61a68eb$</td>
<td>$ac6e3d45$</td>
<td>$752053bb$</td>
<td>$3d471e6d$</td>
<td>$1a564bd$</td>
</tr>
<tr>
<td>4</td>
<td>$e0c879b5$</td>
<td>$e1e83597$</td>
<td>$25bd6b4c$</td>
<td>$d47c6ca11$</td>
<td>$e58d4b5ad$</td>
</tr>
<tr>
<td>5</td>
<td>$9263b1b4$</td>
<td>$4f1cb86c$</td>
<td>$fbcf6b4f$</td>
<td>$d1f3e2f29$</td>
<td>$d180b2b6$</td>
</tr>
<tr>
<td>6</td>
<td>$5a19a37a$</td>
<td>$be4d0a6d$</td>
<td>$be4d0a6d$</td>
<td>$0b515e4a$</td>
<td>$ea5b317f$</td>
</tr>
<tr>
<td>7</td>
<td>$42dc1904$</td>
<td>$2c6d4f2c$</td>
<td>$f2f2c63$</td>
<td>$73baf529$</td>
<td>$f321416e$</td>
</tr>
<tr>
<td>8</td>
<td>$ea046585$</td>
<td>$87f2d49d$</td>
<td>$87f2d49d$</td>
<td>$4740a34c$</td>
<td>$ac192857$</td>
</tr>
<tr>
<td>9</td>
<td>$f02ad4c5$</td>
<td>$8c6d985$</td>
<td>$a68c$</td>
<td>$ed5a6bc$</td>
<td>$f321416e$</td>
</tr>
<tr>
<td>10</td>
<td>$eb59b1b$</td>
<td>$e9cb3d4f$</td>
<td>$e9cb3d4f$</td>
<td>$d0c9e1b6$</td>
<td>$d24d602f$</td>
</tr>
<tr>
<td></td>
<td>$8409850b$</td>
<td>$1dbf9732$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dr. Brian Gladman, v3.16, 1st August 2007 page 21
12. Rijndael Reference Implementation in C++

12.1 The Header File

// Copyright in this code is held by Dr B. R. Gladman but free direct or
d// derivative use is permitted subject to acknowledgement of its origin.
d// There are no guarantees of correctness or fitness for purpose.

#ifndef AES_H
#define AES_H

const int n_row = 4;        // the number or rows in the state
const int n_col = 4;        // the number or columns in the state
const int n_maxr = 14;      // the maximum number of cipher rounds

typedef unsigned char aes_elem;   // a finite field element in GF(256)
typedef aes_elem    aes_col[n_row];     // a column of four GF(256) elements
typedef aes_col    aes_state[n_col];    // an array of columns for the state

class aes
{
    aes_state     key_sch[n_maxr + 1];  // the key schedule
    int           key_len;

    public:
        aes(void) : key_len(0) {};        
        bool key(const void *key, int keylen);  
        bool encrypt(const void* pt, void* ct) const; 
        bool decrypt(const void* ct, void* pt) const;
};
#endif

12.2 The C++ Implementation

// Copyright in this code is held by Dr B. R. Gladman but free direct or
d// derivative use is permitted subject to acknowledgement of its origin.
d// There are no guarantees of correctness or fitness for purpose.

#include "aes.h"

namespace
{
    aes_elem s_box[256] =          // the S box
    {
        0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
        0x30, 0x01, 0x67, 0x2b, 0xfe, 0xdb, 0xa8, 0x85,
        0x04, 0x0c, 0x37, 0x6d, 0x8a, 0x5a, 0x97, 0x9e,
        0xe3, 0x76, 0x8d, 0x94, 0xe1, 0xf9, 0x2f, 0x42,
        0x0f, 0x0c, 0x3c, 0x8d, 0x76, 0x97, 0xf2, 0x71,
        0x7e, 0x39, 0x88, 0x08, 0x40, 0x8b, 0x07, 0x19,
        0xe5, 0x40, 0x4e, 0x5c, 0x6c, 0x8c, 0xe4, 0x5c,
        0x62, 0x0e, 0x8e, 0x5a, 0x0b, 0x1f, 0x53, 0x7d,
        0x09, 0x78, 0xe8, 0x64, 0x76, 0x89, 0x0d, 0x04,
        0xe6, 0x9c, 0x77, 0x8e, 0xa1, 0x85, 0师范, 0x97,
    };
}
\begin{verbatim}
A Specification for The AES Algorithm

\textbf{Rijndael (by Joan Daemen & Vincent Rijmen)}

\textbf{Dr. Brian Gladman}, v3.16, 1st August 2007

page 23

0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c,
0xc2, 0xd3, 0x62, 0x91, 0x95, 0xe4, 0x79,
0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9,
0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
0x7a, 0x2e, 0xc6, 0x6a, 0xbd, 0xe4, 0x5b,
0xe8, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0xe0,
0x7c, 0x97, 0xb9, 0x86, 0xc1, 0x1d, 0xe9,
0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94,
0x9b, 0xe8, 0xe9, 0xce, 0x55, 0x28, 0xdf,
0x8c, 0x1a, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68,
0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16

};

\textbf{aes_elem} is_box[256] = // the inverse S box
{
0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38,
0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,
0x34, 0x08, 0x43, 0xc4, 0xde, 0xe9, 0xcb,
0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d,
0xee, 0x4c, 0x95, 0x0b, 0xfa, 0xc3, 0x4e,
0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2,
0x76, 0x5b, 0xe2, 0xd6, 0x8b, 0xd1, 0x25, 0x62,
0x72, 0xf8, 0xf6, 0x64, 0x86, 0x98, 0x16, 0x03,
0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0x92, 0x26,
0xe0, 0x70, 0x48, 0x50, 0xf7, 0xed, 0xb9, 0xda,
0x5e, 0x15, 0x46, 0xa7, 0x8d, 0x9d, 0x84, 0x1f,
0x90, 0xd8, 0x00, 0xc8, 0xbc, 0xd3, 0x0a,
0x9c, 0x85, 0x4f, 0x8b, 0xb3, 0x45, 0x06, 0x1b,
0xda, 0xe7, 0x35, 0xe6, 0x13, 0x44, 0x7c, 0xe6,
0x97, 0xd8, 0xab, 0x00, 0x8c, 0xe0, 0x1c, 0x6c,
0x9c, 0x5c, 0x6e, 0x77, 0xb9, 0xc7, 0x62, 0x9b,
0x88, 0x11, 0x0e, 0x7b, 0x35, 0xe4, 0x53, 0x08,
0xf7, 0xed, 0x81, 0x99, 0x2a, 0xcc, 0x32, 0x04,
0xad, 0xe8, 0x2f, 0x63, 0x53, 0x43, 0xc9, 0x0f,
0xe3, 0x0d, 0xb5, 0x26, 0x2b, 0x1e, 0x5a, 0xd8,
0x2f, 0x3e, 0x08, 0x52, 0x9e, 0xe9, 0x03, 0x99,
0x76, 0x23, 0x31, 0x9d, 0xad, 0x57, 0x28, 0x47,
0x3c, 0x7f, 0x02, 0x0c, 0x3f, 0x6f, 0xe5, 0x32,
0x60, 0x01, 0x67, 0xe1, 0x6d, 0x31, 0x52, 0x3b,
0x6c, 0x42, 0x51, 0x05, 0x1c, 0xe7, 0x18, 0x1c,
0xff, 0xc7, 0x3d, 0x62, 0x80, 0x52, 0x08, 0x88,
0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16
};

aes_elem ff_tab[8] =
{
0x00, 0x1b, 0x36, 0x2d, 0x6c, 0x77, 0x5a, 0x41
};

// multiply a GF(256) element by \{02\}

aes_elem gf_mul2(aes_elem s)
{
   return (s << 1) ^ ff_tab[s >> 7];
}
\end{verbatim}
// multiply a GF(256) element by {04}

aes_elem gf_mul4(aes_elem s)
{
    return (s << 2) ^ ff_tab[s >> 6];
}

// multiply a GF(256) element by {08}

aes_elem gf_mul8(aes_elem s)
{
    return (s << 3) ^ ff_tab[s >> 5];
}

// rotate bytes within a column

void rot_column(aes_col in)
{
    aes_elem t;
    t = in[0];
    in[0] = in[1];
    in[1] = in[2];
    in[2] = in[3];
    in[3] = t;
}

// forward byte substitution for a column

void sub_column(aes_col in)
{
    in[0] = s_box[in[0]];
    in[1] = s_box[in[1]];
    in[3] = s_box[in[3]];
}

// inverse byte substitution for a column

void inv_sub_column(aes_col in)
{
    in[0] = is_box[in[0]];
    in[1] = is_box[in[1]];
    in[2] = is_box[in[2]];
    in[3] = is_box[in[3]];
}

// forward byte substitution transformation

void sub_bytes(aes_state s)
{
    for(int row = 0; row < n_row; ++row)
        sub_column(s[row]);
}

// inverse byte substitution transformation

void inv_sub_bytes(aes_state s)
{
    for(int row = 0; row < n_row; ++row)
        inv_sub_column(s[row]);
}
// forward shift row transformation
void shift_rows(aes_state s)
{
    for(int row = 1; row < n_row; ++row)
    {
        aes_elem t[n_row];

        t[0] = s[(row + 0) % n_col][row];
        t[1] = s[(row + 1) % n_col][row];
        t[2] = s[(row + 2) % n_col][row];
        t[3] = s[(row + 3) % n_col][row];

        s[0][row] = t[0];
        s[1][row] = t[1];
        s[2][row] = t[2];
        s[3][row] = t[3];
    }
}

// inverse shift row transformation
void inv_shift_rows(aes_state s)
{
    for(int row = 1; row < n_row; ++row)
    {
        aes_elem t[n_row];

        t[(row + 0) % n_col] = s[0][row];
        t[(row + 1) % n_col] = s[1][row];
        t[(row + 2) % n_col] = s[2][row];
        t[(row + 3) % n_col] = s[3][row];

        s[0][row] = t[0];
        s[1][row] = t[1];
        s[2][row] = t[2];
        s[3][row] = t[3];
    }
}

// forward mix column transformation
//  {02}a ^ {03}b ^ {01}c ^ {01}d = a ^ {a ^ b ^ c ^ d} ^ {02}{a ^ b}
//  {01}a ^ {02}b ^ {03}c ^ {01}d = b ^ {a ^ b ^ c ^ d} ^ {02}{b ^ c}
//  {01}a ^ {01}b ^ {02}c ^ {03}d = c ^ {a ^ b ^ c ^ d} ^ {02}{c ^ d}
//  {03}a ^ {01}b ^ {01}c ^ {02}d = d ^ {a ^ b ^ c ^ d} ^ {02}{d ^ a}
void mix_columns(aes_state s)
{
    for(int col = 0; col < n_col; ++col)
    {
        aes_elem ad, bc, abcd;

        ad = s[col][0] ^ s[col][3];
        bc = s[col][1] ^ s[col][2];
        abcd = ad ^ bc;

        s[col][0] ^= abcd ^ gf_mul2(s[col][0] ^ s[col][1]);
        s[col][1] ^= abcd ^ gf_mul2(bc);
        s[col][2] ^= abcd ^ gf_mul2(s[col][2] ^ s[col][3]);
        s[col][3] ^= abcd ^ gf_mul2(ad);
    }
}

// inverse mix column transformation
//  {0e}a ^ {0b}b ^ {0d}c ^ {09}d
void inv_mix_columns(aes_state s)
{
    for (int col = 0; col < n_col; ++col)
    {
        aes_elem ad, bc, p, q;

        ad = s[col][0] ^ s[col][3];
        bc = s[col][1] ^ s[col][2];
        q = ad ^ bc;
        q ^= gf_mul8(q);
        p = q ^ gf_mul4(s[col][0] ^ s[col][2]);
        q = q ^ gf_mul4(s[col][1] ^ s[col][3]);

        s[col][0] ^= p ^ gf_mul2(s[col][0] ^ s[col][1]);
        s[col][1] ^= q ^ gf_mul2(bc);
        s[col][2] ^= p ^ gf_mul2(s[col][2] ^ s[col][3]);
        s[col][3] ^= q ^ gf_mul2(ad);
    }
}

// add key transformation (same in both directions)

void add_round_key(aes_state s, const aes_state k)
{
    for (int col = 0; col < n_col; ++col)
    {
        for (int row = 0; row < n_row; ++row)
            s[col][row] ^= k[col][row];
    }
}

// transfer the input to the state array

void state_in(aes_state s, const void* in)
{
    for (int col = 0, i = 0; col < n_col; ++col)
        for (int row = 0; row < n_row; ++row)
            s[col][row] = static_cast<const unsigned char*>(in)[i++];
}

// transfer the state array to the output

void state_out(aes_state s, void* out)
{
    for (int col = 0, i = 0; col < n_col; ++col)
        for (int row = 0; row < n_row; ++row)
            static_cast<unsigned char*>(out)[i++] = s[col][row];
}
// set the cipher key

bool aes::key(const void* key, int keylen) {
    if(keylen == 128 || keylen == 192 || keylen == 256)
        key_len = keylen / 32;
    else if(keylen == 16 || keylen == 24 || keylen == 32)
        key_len = key_len / 4;
    else
        return false;

    aes_col *kp = (aes_col*)key_sch;
    int hi = n_col * (key_len + 7);
    int i = -1, k = 0;

    while(++i < key_len)
    {
        kp[i][0] = static_cast<const unsigned char*>(key)[k++];
        kp[i][1] = static_cast<const unsigned char*>(key)[k++];
        kp[i][2] = static_cast<const unsigned char*>(key)[k++];
        kp[i][3] = static_cast<const unsigned char*>(key)[k++];
    }

    --i;
    aes_elem rc = 1;

    while(++i < hi)
    {
        aes_col temp = { kp[i - 1][0], kp[i - 1][1],
                        kp[i - 1][2], kp[i - 1][3] };

        if(i % key_len == 0)
        {
            rot_column(temp);
            sub_column(temp);
            temp[0] ^= rc;
            rc = gf_mul2(rc);
        }
        else if((key_len > 6 && (i % key_len == 4))
                 sub_column(temp);
        }

        kp[i][0] = kp[i - key_len][0] ^ temp[0];
        kp[i][1] = kp[i - key_len][1] ^ temp[1];
        kp[i][2] = kp[i - key_len][2] ^ temp[2];
        kp[i][3] = kp[i - key_len][3] ^ temp[3];
    }

    return true;
}

// encrypt a single block of 16 bytes

bool aes::encrypt(const void *pt, void* ct) const
{ aes_state s;

    if(key_len)
    {
        state_in(s, pt);
        add_round_key(s, key_sch[0]);

        static_cast<unsigned char*>(out)[i++] = s[col][row];
    }
for(int r = 1; r < key_len + 6; ++r)
{
    sub_bytes(s);
    shift_rows(s);
    mix_columns(s);
    add_round_key(s, key_sch[r]);
}

sub_bytes(s);
shift_rows(s);
add_round_key(s, key_sch[key_len + 6]);

state_out(s, ct);
return true;
else
    return false;
}

// decrypt a single block of 16 bytes

bool aes::decrypt(const void* ct, void* pt) const
{
    aes_state s;

    if(key_len)
    {
        state_in(s, ct);
        add_round_key(s, key_sch[key_len + 6]);

        for(int r = key_len + 5; r > 0; --r)
        {
            inv_shift_rows(s);
            inv_sub_bytes(s);
            add_round_key(s, key_sch[r]);
            inv_mix_columns(s);
        }

        inv_shift_rows(s);
        inv_sub_bytes(s);
        add_round_key(s, key_sch[0]);

        state_out(s, pt);
        return true;
    }
    else
        return false;
}

13. Rijndael Development Test Vectors

All vectors are in hexadecimal notation with each pair of characters giving a byte value where the left and right characters of each pair provide the bit pattern for the 4 bit group containing the higher and lower numbered bits respectively using the format explained in Section 1.2. The array index for all bytes (groups of two hexadecimal digits) within these test vectors starts at zero on the left and increases from left to right.

Considered instead as bit sequences, with hexadecimal digits numbered from left to right starting from 0, hexadecimal digit \( n \) gives the value of bits \( 4n \) to \( 4n + 3 \) in the sequence using the 4-bit notation given in Section 1.2 except that lower numbered bits are now on the left (this arises because bits in bit sequences and bits in bytes are mapped in reverse).
These test have been generated by Dr Brian Gladman using the 
program aes_vec.cpp  <brg@gladman.uk.net> 24th January 2001.

## LEGEND FOR ENCRYPT (round number r = 0 to 10, 12 or 14)

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>state at start of round[r]</td>
</tr>
<tr>
<td>s_box</td>
<td>state after s_box substitution</td>
</tr>
<tr>
<td>s_row</td>
<td>state after shift row transformation</td>
</tr>
<tr>
<td>m_col</td>
<td>state after mix column transformation</td>
</tr>
<tr>
<td>k_sch</td>
<td>key schedule value for round[r]</td>
</tr>
<tr>
<td>output</td>
<td>cipher output</td>
</tr>
</tbody>
</table>

## LEGEND FOR DECRYPT (round number r = 0 to 10, 12 or 14)

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>state at start of round[r]</td>
</tr>
<tr>
<td>is_box</td>
<td>state after inverse s_box substitution</td>
</tr>
<tr>
<td>is_row</td>
<td>state after inverse shift row transformation</td>
</tr>
<tr>
<td>ik_sch</td>
<td>key schedule value for round[r]</td>
</tr>
<tr>
<td>ik_add</td>
<td>state after key addition</td>
</tr>
<tr>
<td>ioutput</td>
<td>cipher output</td>
</tr>
</tbody>
</table>

## PLANETEXT: 3243f6a8885a308d313198a2e0370734 (pi * 2^124)

## KEY: 2b7e151628aed2a6abf7158809cf4f3c ( e * 2^124)

### ENCRYPT

<table>
<thead>
<tr>
<th>Round</th>
<th>input</th>
<th>0x3243f6a8885a308d313198a2e0370734</th>
<th>0x2b7e151628aed2a6abf7158809cf4f3c</th>
</tr>
</thead>
<tbody>
<tr>
<td>R[0]</td>
<td>k_sch</td>
<td>0x2b7e151628aed2a6abf7158809cf4f3c</td>
<td></td>
</tr>
<tr>
<td>R[1]</td>
<td>start</td>
<td>0x193de3be0a4f7ae08ac68d2a9f84803</td>
<td></td>
</tr>
<tr>
<td>R[2]</td>
<td>s_box</td>
<td>0xd42711aee0bf98f1b88b45de51e415230</td>
<td></td>
</tr>
<tr>
<td>R[3]</td>
<td>s_row</td>
<td>0xd4bf5d30e0b452aeb8411f1e2798e5</td>
<td></td>
</tr>
<tr>
<td>R[4]</td>
<td>m_col</td>
<td>0x046681e5e0ca1b9a948f8d37a28062d4e</td>
<td></td>
</tr>
<tr>
<td>R[5]</td>
<td>k_sch</td>
<td>0xa0fafe17885b98f1b8b452aeb8411f1e2798e5</td>
<td></td>
</tr>
<tr>
<td>R[6]</td>
<td>k_sch</td>
<td>0x49de2d4f965f972b98f1b8b452aeb8411f1e2798e5</td>
<td></td>
</tr>
<tr>
<td>R[7]</td>
<td>k_sch</td>
<td>0xc0524281e9f897b352aeb8411f1e2798e5</td>
<td></td>
</tr>
</tbody>
</table>

### DECRIPT (MOD) (round number r = 0 to 10, 12 or 14)

<table>
<thead>
<tr>
<th>Round</th>
<th>input</th>
<th>0x3243f6a8885a308d313198a2e0370734</th>
<th>0x2b7e151628aed2a6abf7158809cf4f3c</th>
</tr>
</thead>
<tbody>
<tr>
<td>R[0]</td>
<td>k_sch</td>
<td>0x2b7e151628aed2a6abf7158809cf4f3c</td>
<td></td>
</tr>
<tr>
<td>R[1]</td>
<td>start</td>
<td>0x193de3be0a4f7ae08ac68d2a9f84803</td>
<td></td>
</tr>
<tr>
<td>R[2]</td>
<td>s_box</td>
<td>0xd42711aee0bf98f1b88b45de51e415230</td>
<td></td>
</tr>
<tr>
<td>R[3]</td>
<td>s_row</td>
<td>0xd4bf5d30e0b452aeb8411f1e2798e5</td>
<td></td>
</tr>
<tr>
<td>R[4]</td>
<td>m_col</td>
<td>0x046681e5e0ca1b9a948f8d37a28062d4e</td>
<td></td>
</tr>
<tr>
<td>R[5]</td>
<td>k_sch</td>
<td>0xa0fafe17885b98f1b8b452aeb8411f1e2798e5</td>
<td></td>
</tr>
<tr>
<td>R[6]</td>
<td>k_sch</td>
<td>0x49de2d4f965f972b98f1b8b452aeb8411f1e2798e5</td>
<td></td>
</tr>
<tr>
<td>R[7]</td>
<td>k_sch</td>
<td>0xc0524281e9f897b352aeb8411f1e2798e5</td>
<td></td>
</tr>
</tbody>
</table>

### KEY SCHEDULE FOR INVERSE MIX COLUMN FOLLOWED BY KEY XOR

<table>
<thead>
<tr>
<th>Round</th>
<th>input</th>
<th>0x3243f6a8885a308d313198a2e0370734</th>
<th>0x2b7e151628aed2a6abf7158809cf4f3c</th>
</tr>
</thead>
<tbody>
<tr>
<td>R[0]</td>
<td>k_sch</td>
<td>0x2b7e151628aed2a6abf7158809cf4f3c</td>
<td></td>
</tr>
<tr>
<td>R[1]</td>
<td>start</td>
<td>0x193de3be0a4f7ae08ac68d2a9f84803</td>
<td></td>
</tr>
<tr>
<td>R[2]</td>
<td>s_box</td>
<td>0xd42711aee0bf98f1b88b45de51e415230</td>
<td></td>
</tr>
<tr>
<td>R[3]</td>
<td>s_row</td>
<td>0xd4bf5d30e0b452aeb8411f1e2798e5</td>
<td></td>
</tr>
<tr>
<td>R[4]</td>
<td>m_col</td>
<td>0x046681e5e0ca1b9a948f8d37a28062d4e</td>
<td></td>
</tr>
<tr>
<td>R[5]</td>
<td>k_sch</td>
<td>0xa0fafe17885b98f1b8b452aeb8411f1e2798e5</td>
<td></td>
</tr>
<tr>
<td>R[6]</td>
<td>k_sch</td>
<td>0x49de2d4f965f972b98f1b8b452aeb8411f1e2798e5</td>
<td></td>
</tr>
<tr>
<td>R[7]</td>
<td>k_sch</td>
<td>0xc0524281e9f897b352aeb8411f1e2798e5</td>
<td></td>
</tr>
</tbody>
</table>
\[ R[7].s_row \quad \text{f783403f27433df09bb531ff54aba9d3} \]
\[ R[7].m_col \quad 1415b5bf46165ec724656d7342ad843 \]
\[ R[7].k_sch \quad 4e54f70e5f5ff9c9384a6f24a6edc4df \]
\[ R[8].start \quad 5a4142b1b1949d1ca3e019567a8c040c \]
\[ R[8].s_box \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[8].m_col \quad \text{be3d4fedef4e1f2ce8a0442cc0aa83864d} \]
\[ R[8].k_sch \quad \text{00512fd1b1c889ff54766dcd6afa199ea} \]
\[ R[9].s_row \quad \text{e2d7321b58b8da2312bf5607f8d922f} \]
\[ R[9].m_col \quad \text{ea835cf00445332d65d98ad8596b0c5} \]
\[ R[9].k_sch \quad \text{87ec4a8cf26ec46959790e7a6} \]
\[ R[10].start \quad \text{5a4142b11949dc1fa3e019657a8c040c} \]
\[ R[10].s_box \quad \text{be832cc8d43b86c00aed4dada6f2fe} \]
\[ R[10].m_col \quad \text{be3d4fedef4e1f2ce8a0442cc0aa83864d} \]
\[ R[10].k_sch \quad \text{00512fd1b1c889ff54766dcd6afa199ea} \]

DECYPRT
\[ R[0].i_input \quad \text{3925841d02dc09fbdcc18597196a0b32} \]
\[ R[1].i_k_sch \quad \text{d014f9a8c9ee2589e13f0cc8b6630ca6} \]
\[ R[1].i_start \quad \text{e9317b5cb322c723d2de895fa090794} \]
\[ R[1].i_box \quad \text{f783403f27433df09bb531ff54aba9d3} \]
\[ R[1].i_k_sch \quad \text{e9317b5cb322c723d2de895fa090794} \]
\[ R[1].i_k_add \quad \text{4e54f70e5f5ff9c9384a6f24a6edc4df} \]
\[ R[1].i_box \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[1].i_k_sch \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[1].i_k_add \quad \text{4e54f70e5f5ff9c9384a6f24a6edc4df} \]
\[ R[2].i_start \quad \text{f783403f27433df09bb531ff54aba9d3} \]
\[ R[2].i_box \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[2].i_k_sch \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[2].i_k_add \quad \text{4e54f70e5f5ff9c9384a6f24a6edc4df} \]
\[ R[2].i_box \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[2].i_k_sch \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[2].i_k_add \quad \text{4e54f70e5f5ff9c9384a6f24a6edc4df} \]
\[ R[3].i_start \quad \text{f783403f27433df09bb531ff54aba9d3} \]
\[ R[3].i_box \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[3].i_k_sch \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[3].i_k_add \quad \text{4e54f70e5f5ff9c9384a6f24a6edc4df} \]
\[ R[3].i_box \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[3].i_k_sch \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[3].i_k_add \quad \text{4e54f70e5f5ff9c9384a6f24a6edc4df} \]
\[ R[4].i_start \quad \text{f783403f27433df09bb531ff54aba9d3} \]
\[ R[4].i_box \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[4].i_k_sch \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[4].i_k_add \quad \text{4e54f70e5f5ff9c9384a6f24a6edc4df} \]
\[ R[4].i_box \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[4].i_k_sch \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[4].i_k_add \quad \text{4e54f70e5f5ff9c9384a6f24a6edc4df} \]
\[ R[5].i_start \quad \text{f783403f27433df09bb531ff54aba9d3} \]
\[ R[5].i_box \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[5].i_k_sch \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[5].i_k_add \quad \text{4e54f70e5f5ff9c9384a6f24a6edc4df} \]
\[ R[5].i_box \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[5].i_k_sch \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[5].i_k_add \quad \text{4e54f70e5f5ff9c9384a6f24a6edc4df} \]
\[ R[6].i_start \quad \text{f783403f27433df09bb531ff54aba9d3} \]
\[ R[6].i_box \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[6].i_k_sch \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[6].i_k_add \quad \text{4e54f70e5f5ff9c9384a6f24a6edc4df} \]
\[ R[6].i_box \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[6].i_k_sch \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[6].i_k_add \quad \text{4e54f70e5f5ff9c9384a6f24a6edc4df} \]
\[ R[7].i_start \quad \text{f783403f27433df09bb531ff54aba9d3} \]
\[ R[7].i_box \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[7].i_k_sch \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[7].i_k_add \quad \text{4e54f70e5f5ff9c9384a6f24a6edc4df} \]
\[ R[7].i_box \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[7].i_k_sch \quad \text{be3832cc8d43b86c00aed4dada6f2fe} \]
\[ R[7].i_k_add \quad \text{4e54f70e5f5ff9c9384a6f24a6edc4df} \]

Dr. Brian Gladman, v3.16, 1st August 2007
R[10].output 3243f6a8885a308d313198a2e0370734
DECRIPT (MOD)
R[ 0].input 3925841d02dc09fbdc11859716a0b32
R[ 0].k_sch d014f9a8c9ee2589e13f0cc8b6630ca6
R[ 1].start e9317dbb5c322c723d2e895fafa097094
R[ 1].is_box eb2e13d259a1421e8bc3f2841b4038e7
R[ 1].is_row eb40f21e592e38848ba1137bc342d2
R[ 1].im_col 8b151cc5e1550d72fda9c248faa30821
R[ 1].k_sch 0c7b5a631319eafeb039898064fcbf4
R[ 2].start 876e46a6f2a4e78cd404ad979ec9935
R[ 2].is_box ea598c5045d0f065965ce2d8553334ad
R[ 2].is_row ea835cf0445332d6559d9a8596b0c9
R[ 2].k_sch 0f44df5646a4c834255a9444ae0efce6569
R[ 2].istart df7d925a1f6b209d36a032626ed675732
R[ 3].start 0f3b3df44f1ef2c80a642cc00da83864d
R[ 3].is_box 5a49190c19e004b1a38c21f7a141d6c5
R[ 3].is_col 762e7160f38b4da5762e7160f38b4da5
R[ 4].start 0f3b3df44f1ef2c80a642cc00da83864d
R[ 4].is_box 5a49190c19e004b1a38c21f7a141d6c5
R[ 4].is_row 5a49190c19e004b1a38c21f7a141d6c5
R[ 4].im_col 614646a4c834255a9444ae0efce6569
R[ 4].k_sch df7d925a1f6b209d36a032626ed675732
R[ 5].start 0f3b3df44f1ef2c80a642cc00da83864d
R[ 5].is_box 5a49190c19e004b1a38c21f7a141d6c5
R[ 5].is_row 5a49190c19e004b1a38c21f7a141d6c5
R[ 5].im_col 614646a4c834255a9444ae0efce6569
R[ 5].k_sch df7d925a1f6b209d36a032626ed675732
R[ 6].start 0f3b3df44f1ef2c80a642cc00da83864d
R[ 6].is_box 5a49190c19e004b1a38c21f7a141d6c5
R[ 6].is_row 5a49190c19e004b1a38c21f7a141d6c5
R[ 6].im_col 614646a4c834255a9444ae0efce6569
R[ 6].k_sch df7d925a1f6b209d36a032626ed675732
R[ 7].start 0f3b3df44f1ef2c80a642cc00da83864d
R[ 7].is_box 5a49190c19e004b1a38c21f7a141d6c5
R[ 7].is_row 5a49190c19e004b1a38c21f7a141d6c5
R[ 7].im_col 614646a4c834255a9444ae0efce6569
R[ 7].k_sch df7d925a1f6b209d36a032626ed675732
R[ 8].start 0f3b3df44f1ef2c80a642cc00da83864d
R[ 8].is_box 5a49190c19e004b1a38c21f7a141d6c5
R[ 8].is_row 5a49190c19e004b1a38c21f7a141d6c5
R[ 8].im_col 614646a4c834255a9444ae0efce6569
R[ 8].k_sch df7d925a1f6b209d36a032626ed675732
R[ 9].start 0f3b3df44f1ef2c80a642cc00da83864d
R[ 9].is_box 5a49190c19e004b1a38c21f7a141d6c5
R[ 9].is_row 5a49190c19e004b1a38c21f7a141d6c5
R[ 9].im_col 614646a4c834255a9444ae0efce6569
R[ 9].k_sch df7d925a1f6b209d36a032626ed675732
R[10].istart 0f3b3df44f1ef2c80a642cc00da83864d
R[10].k_sch 0f3b3df44f1ef2c80a642cc00da83864d
R[10].is_box 5a49190c19e004b1a38c21f7a141d6c5
R[10].is_row 5a49190c19e004b1a38c21f7a141d6c5
R[10].im_col 614646a4c834255a9444ae0efce6569
R[10].istart 0f3b3df44f1ef2c80a642cc00da83864d
R[10].k_sch 0f3b3df44f1ef2c80a642cc00da83864d
R[10].is_box 5a49190c19e004b1a38c21f7a141d6c5
R[10].is_row 5a49190c19e004b1a38c21f7a141d6c5
R[10].im_col 614646a4c834255a9444ae0efce6569

PLAINTEXT: 3243f6a8885a308d313198a2e0370734

KEY: 2b7e151628a26aaf7158809c4f4f3c

ENCRYPT
R[ 0].input 3243f6a8885a308d313198a2e0370734
R[ 0].k_sch 2b7e151628a26aaf7158809c4f4f3c
R[ 1].start 193e3bea0f4e22b9a6862bae9f84808
R[ 1].s_box 40528ec977d092075c4df4e6eef09694a1
R[ 1].s_row 4009ba17d4d947c9f6e8c75f052200ef
R[ 2].m_col 8026de2a2ed7a16bddd2c5bacc2bdc8bc
R[4].is_row 223a2ec75534081d0ef6a6ecb3d35572
R[4].is_box 94a2c331ed28bfded7d6c5834ba9ed1e
R[4].i_input 5001974ea3f890f7435b36c5b27bba3f9e
R[4].ik_add 24a3547f4fd1b720a3e37b3f19d25780
R[5].istart 51b0622fe4b50e564ab19a654b6c3d06
R[5].is_row 516c9a56e4b03d654ab562064bb10e2f
R[5].is_box 70b837b9ae8bc45c2daaba5cc565d74e
R[5].ik_sch d6ccb42624e04229f1054d54ad51d52
R[5].ik_add a67481fb8b82f9ee3c2f708683ca1c
R[6].istart 1a076d48810d96a3d49861b64a4766b
R[6].is_row 1a4a968a8104761a3107668b6d49d49d
R[6].is_box 435ce3b8977c16d87170cf79791e51f
R[6].ik_sch a0e27c22d5c54987fad48a1c2f89f7b0
R[6].ik_add e3be257a429f55f88eb88eb6be17aa9
R[7].istart 7ebcbbc7cb9949b2be6f7622f875024aa
R[7].is_row 7e5072b2b9bc242fe69bc2aaf786497c
R[7].is_box 8a6c13e3eb78a64ef5db7862ead6a401
R[7].ik_sch 2f11c39be82c15acc43492f2f082b260
R[7].ik_add a57dda5335a3b32e313ef15901a51d61f
R[8].istart 22be42856b2c229eaf201a95c88ba3f
R[8].is_row 9499c6ee9789412bb0409b7a797c025
R[8].is_box 2c183c5e34b699b2bd517ae75278e5a
R[8].ik_add b881fab08de00f8001d19d2b04e580
R[9].istart 1f755ba2c283e24fdf6554a1eb2c345f
R[9].is_row 1f2c544fc375341a0f82bb5fe65c2a2e
R[9].is_box cb42fd9233f284311f8e43bcfa81a
R[9].ik_sch f1c158b6ce82990ba5636d3e373d663a
R[9].ik_add 3a83a524fcdcbf1487b273b3abf817e2d
R[10].istart fa8615516dca8c00596d7eaa5798066
R[10].is_row fa9867c06d880aa05cc5166579d85a1
R[10].is_box 14e20a1fb3dca3a6236272fd3a756f70
R[10].ik_sch 94cd4359d0b909eb25ea6918aaa7cc
R[10].ik_add 8026de2a2ed7a16bddd02c5bac2dbcb8c
R[11].istart 4009baa17d49d497cf968c75f05220eef
R[11].is_row 40528c977d92075fcf4daaeef09649a1
R[11].is_box 7248f0851340543f5f65c061735e7f1
R[11].ik_sch 762e7160f38b4da5179d313b3f3c31bd
R[11].ik_add 046681e5e0cb199a48f83d7a2806264c
R[12].istart d4bf5d30e0b452aeb8411f11e2798e5
R[12].is_row d42711ae0bf98f1b8b45de51e415230
R[12].is_box 193de3bea0f4e22b9a68d82a9ef48f808
R[12].ik_sch 2b7e151268aed2a6abf715880c9f43f3c
R[12].ioutput 3243f6a8885a308d31319b2ae0370734
DECRYPT (MOD)
R[0].i_input f9fb29aefc384a2503408d383b78ebc00
R[0].ik_sch e3936061fe7324d287a36b5a833baf9e5
R[1].istart 1a689cf024b6ef784e369db80c454e5
R[1].is_box 43f7a45ffacc45264f4db3e83a8862a
R[1].is_row 4388b3266af768e84f3cca42a3a44d5f5
R[1].im_col be8f7fa65b8e7eb6c47301b3dbee5f
R[1].ik_sch d9a8b586e110471147e6fe092e5dd0
R[2].istart 6240427c57bc3f66f12320fa1231e325
R[2].is_row ab72ff01dacc12062b32540c39824dc2
R[2].is_box ab825406da7240c2bcb6c639321201
R[2].im_col a69afef1b0f45b8b691d6225b660405
R[2].ik_sch a6ff20df4e13a1e89bc120dd9e94f9
R[3].istart 00628f3055ce7fa63f2dc42f828d8d49
R[3].is_box 52a87308edec6bc525fa884e112191c
R[3].is_row 52d288c5eda19e25ec713c11fa6b08
R[3].im_col 5048eecc05f4d3330ec50f62882af79e
R[3].ik_sch 727c48b6500286f400162173f8b297
R[4].istart 2234a6725f655c70ed32e1db33a08e
R[4].is_row 9428c51e3ed6ed317a9c3de4aa2babf3
R[4].is_box 94a2c331ed28bfded7d6c5834ba9ed1e
R[4].im_col 01a4c5aad1cbbdbed4f6482a35227c7
R[5].istart 5014a78b38ae3e89e47d248e83e1ac1
R[5].is_row 51b0622fe4b50e564ab19a654b6c3d06
R[5].is_box 70fcb4ea2d27b95c5637bcccbb8ba5
R[5].is_row 70b837b9ae68bbc5cd2aaba5cc56d74e
A Specification for The AES Algorithm  
Rijndael (by Joan Daemen & Vincent Rijmen) 

Dr. Brian Gladman, v3.16, 1st August 2007  

page 34
| R[6].s_box | 8ae6f21acca838f0c6b5e4eebfcf |
| R[6].s_row | 8aa854cfc6b63f3a06eeef2f4ee63821 |
| R[6].k_sch | 77f2c0f6f7e397a263008f5b4f00b6eb6e |
| R[6].start | 14611694d2b1173569b87fc2a6a7315f |
| R[7].s_box | 6393d6692da2e4f4288777312758a11 |
| R[7].s_row | fbdce45d85731842cc4f59ac957b82 |
| R[7].m_col | 77f2c0f6f7e397a263008f5b4f00b6eb6e |
| R[8].start | 2926ae58f432234bf070ff6569e4423 |
| R[8].s_box | 5f7e46af23263b038c511697b10b1b26 |
| R[8].s_row | 5f231626bf11596a80be4db31b7269f9 |
| R[8].m_col | 6393d6692da2e4f4288777312758a11 |
| R[9].start | 3sfda4302e0596e3932655e43adcb2 |
| R[9].s_box | 75576923ab8bfc8dd3f6c6980bdf1 |
| R[9].s_row | 6393d6692da2e4f4288777312758a11 |
| R[9].m_col | 6393d6692da2e4f4288777312758a11 |
| R[10].start | 4dfcc08de75a64494e9a1dc2b1b03c |
| R[10].s_box | 3b25242bfe994338a802e4c482687b1d7 |
| R[10].s_row | 3b25242bfe994338a802e4c482687b1d7 |
| R[10].m_col | 3b25242bfe994338a802e4c482687b1d7 |
| R[11].start | 5b9e6b21cc583f642769b824c72f3f |
| R[11].s_box | 7f769b824c72f3f |