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:

\[
b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x + b_0 = \sum_{i=0}^{7} b_i x^i \quad (1.2.1)
\]

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 \( \{01100011\} \) 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 \( \{01100011\} \) 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 \( \{11b\} \). 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 \( Nc \) columns, where \( Nc \) 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 - Input to, and output from, the cipher state array](image)

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:

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

(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:

\[
(x^5 + x^4 + x^2 + x + 1) + (x^7 + x + 1) \equiv (x^7 + x^6 + x^4 + x^2) \quad \text{(Polynomial)}
\]

\[
\{01010111\} \oplus \{10000011\} \equiv \{11010100\} \quad \text{(Binary)}
\]

\[
\{57\} \oplus \{83\} \equiv \{d4\} \quad \text{(Hexadecimal)}
\]
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:

$$m(x) = x^8 + x^4 + x^3 + x + 1$$  \hfill (2.2.1)

Since this polynomial has powers of $x$ up to 8 it cannot be represented by a single byte and will be written as either $1\{0011011\}$ or $1\{1b\}$ as indicated earlier. This process is illustrated in the following example of the product $\{57\} \cdot \{83\} = \{c1\}$ (where $\cdot$ is used to represent finite field multiplication):

$$\left( x^6 + x^4 + x^2 + x + 1 \right) \cdot \left( x^7 + x + 1 \right) \Rightarrow$$

$$\begin{align*}
(x^6 + x^4 + x^2 + x + 1) \cdot x^7 &= x^{13} + x^{11} + x^9 + x^8 + x^7 \\
(x^6 + x^4 + x^2 + x + 1) \cdot x &= x^7 + x^5 + x^3 + x^2 + x \\
(x^6 + x^4 + x^2 + x + 1) \cdot 1 &= \frac{x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1}{x^6 + x^4 + x^2 + x + 1} \\
\end{align*}$$

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

$$\begin{align*}
(x^8 + x^4 + x^3 + x + 1) \cdot x^5 &= \frac{x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1}{x^9 + x^8 + x^5 + x^3 + 1} \\
subtract to give intermediate value &\quad \frac{x^{11} + x^7 + x^6 + x^4 + x^3 + 1}{x^9 + x^8 + x^5 + x^3 + 1} \\
(x^8 + x^4 + x^3 + x + 1) \cdot x^3 &= \frac{x^{11} + x^7 + x^6 + x^4 + x^3 + 1}{x^7 + x^6 + x^4 + x^3 + 1} \\
subtract to give the final remainder &\quad \frac{x^3}{x^7 + x^6 + x^4 + x^3 + 1}
\end{align*}$$

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:

$$b(x) \cdot a(x) \oplus m(x) \cdot c(x) = 1 \quad b(x) \cdot a(x) \mod m(x) = 1 \quad (2.2.2)$$

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

$$a(x) \cdot (b(x) \oplus c(x)) = a(x) \cdot b(x) \oplus a(x) \cdot c(x) \quad (2.2.3)$$

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 GF(256).

2.3 Multiplication by Repeated Shifts

The finite field element $\{00000010\}$ 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 $\{11001000\}$ by $x$ gives an initial result is $1\{10010000\}$. The ‘overflow’ bit is then removed by adding $1\{00011011\}$, the modular polynomial, using an exclusive-or operation to give a final result of $\{10001011\}$.

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>( \Theta \ m(x) )</th>
<th>( {57} \cdot x^p )</th>
<th>bit ( p ) of ( {83} \rightarrow \Theta )</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{01010111}</td>
<td>{01010111}</td>
<td>1</td>
<td>{01010111}</td>
<td>{01010111}</td>
</tr>
<tr>
<td>1</td>
<td>{10101110}</td>
<td>{10101110}</td>
<td>1</td>
<td>{10101110}</td>
<td>{11111101}</td>
</tr>
<tr>
<td>2</td>
<td>{101011100}</td>
<td>{00011101}</td>
<td>{01000111}</td>
<td>0</td>
<td>{00000000}</td>
</tr>
<tr>
<td>3</td>
<td>{100011110}</td>
<td>{10001111}</td>
<td>0</td>
<td>{00000000}</td>
<td>{11111101}</td>
</tr>
<tr>
<td>4</td>
<td>{1000111100}</td>
<td>{00001110}</td>
<td>{00000000}</td>
<td>0</td>
<td>{00000000}</td>
</tr>
<tr>
<td>5</td>
<td>{000111110}</td>
<td>{00011110}</td>
<td>0</td>
<td>{00000000}</td>
<td>{11111101}</td>
</tr>
<tr>
<td>6</td>
<td>{000111100}</td>
<td>{00011110}</td>
<td>0</td>
<td>{00000000}</td>
<td>{11111101}</td>
</tr>
<tr>
<td>7</td>
<td>{0001111000}</td>
<td>{00011100}</td>
<td>1</td>
<td>{00000000}</td>
<td>{11000001}</td>
</tr>
</tbody>
</table>

Table 1 – Finite field multiply \{57\} • \{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 \( a = g^a \) and \( b = g^b \) have the product \( a \cdot b = g^{a+b} \). With a ‘logarithm’ table listing the power of the generator for each finite field element we can hence find the powers \( a \) and \( b \) 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.

<table>
<thead>
<tr>
<th>( L(x,y) )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>{00}</td>
</tr>
<tr>
<td>1</td>
<td>{04}</td>
</tr>
<tr>
<td>2</td>
<td>{07}</td>
</tr>
<tr>
<td>3</td>
<td>{06}</td>
</tr>
<tr>
<td>4</td>
<td>{96}</td>
</tr>
<tr>
<td>5</td>
<td>{66}</td>
</tr>
<tr>
<td>6</td>
<td>{7e}</td>
</tr>
<tr>
<td>7</td>
<td>{2b}</td>
</tr>
<tr>
<td>8</td>
<td>{af}</td>
</tr>
<tr>
<td>9</td>
<td>{2c}</td>
</tr>
<tr>
<td>a</td>
<td>{7f}</td>
</tr>
<tr>
<td>b</td>
<td>{cc}</td>
</tr>
<tr>
<td>c</td>
<td>{97}</td>
</tr>
<tr>
<td>d</td>
<td>{53}</td>
</tr>
<tr>
<td>e</td>
<td>{44}</td>
</tr>
<tr>
<td>f</td>
<td>{67}</td>
</tr>
</tbody>
</table>

Table 2 – ‘Logs’ – \( L \) values such that \( (xy) = \{03\}^L \) for a given a finite field element \( (xy) \)
<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>E(xy)</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00 03 05</td>
<td>0f 11</td>
<td>33</td>
<td>55</td>
<td>1a</td>
<td>2e</td>
<td>72</td>
<td>96</td>
</tr>
<tr>
<td>1</td>
<td>5f 01</td>
<td>38 4b</td>
<td>6d</td>
<td>73</td>
<td>95</td>
<td>a4</td>
<td>f7</td>
<td>02</td>
</tr>
<tr>
<td>2</td>
<td>e5 34</td>
<td>5c 4e</td>
<td>37</td>
<td>59</td>
<td>eb</td>
<td>2e</td>
<td>6a</td>
<td>be</td>
</tr>
<tr>
<td>3</td>
<td>53 f5</td>
<td>04 0c</td>
<td>14</td>
<td>3c</td>
<td>44</td>
<td>cc</td>
<td>4f</td>
<td>d1</td>
</tr>
<tr>
<td>4</td>
<td>4c d4</td>
<td>67 a9</td>
<td>e0</td>
<td>3b</td>
<td>4d</td>
<td>d7</td>
<td>62</td>
<td>a6</td>
</tr>
<tr>
<td>5</td>
<td>83 9e</td>
<td>b9 d0</td>
<td>6b</td>
<td>bd</td>
<td>dc</td>
<td>7f</td>
<td>81</td>
<td>b3</td>
</tr>
<tr>
<td>6</td>
<td>b5 c4</td>
<td>57 f9</td>
<td>10</td>
<td>30</td>
<td>50</td>
<td>f0</td>
<td>0b</td>
<td>1d</td>
</tr>
<tr>
<td>7</td>
<td>fe 19</td>
<td>2b 7d</td>
<td>87</td>
<td>92</td>
<td>ad</td>
<td>ec</td>
<td>2f</td>
<td>71</td>
</tr>
<tr>
<td>8</td>
<td>fb 16</td>
<td>3a 4e</td>
<td>d2</td>
<td>6d</td>
<td>b7</td>
<td>c2</td>
<td>5d</td>
<td>e7</td>
</tr>
<tr>
<td>9</td>
<td>c3 5e</td>
<td>e2 3d</td>
<td>47</td>
<td>c9</td>
<td>40</td>
<td>c0</td>
<td>5b</td>
<td>ed</td>
</tr>
<tr>
<td></td>
<td>a9 1f</td>
<td>21 63</td>
<td>a5</td>
<td>f4</td>
<td>07</td>
<td>09</td>
<td>1b</td>
<td>2d</td>
</tr>
<tr>
<td></td>
<td>fb 4f</td>
<td>6a 8b</td>
<td>96</td>
<td>91</td>
<td>a8</td>
<td>e3</td>
<td>3e</td>
<td>4c</td>
</tr>
<tr>
<td></td>
<td>a5 6b</td>
<td>c1 58</td>
<td>e8</td>
<td>23</td>
<td>65</td>
<td>af</td>
<td>ea</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>b9 c7</td>
<td>52 9c</td>
<td>8a</td>
<td>87</td>
<td>94</td>
<td>a7</td>
<td>f2</td>
<td>9d</td>
</tr>
<tr>
<td></td>
<td>e4 1c</td>
<td>9a 58</td>
<td>19</td>
<td>6b</td>
<td>86</td>
<td>8b</td>
<td>98</td>
<td>9b</td>
</tr>
<tr>
<td></td>
<td>c0 4d</td>
<td>7b 8d</td>
<td>2b</td>
<td>97</td>
<td>a2</td>
<td>fd</td>
<td>1c</td>
<td>2a</td>
</tr>
</tbody>
</table>

Table 3 - 'Antilogs' - field elements \(E\) such that \(E = \{03\}^{(x)}\) given the power \(xy\)

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 \(\{a1\}\), as before. These tables can also be used to find the inverses of field elements since \(g^{(x)}\) has the inverse \(g^{(57)}\). Hence the element \(\{af\} = \{03\}^{b7}\) has the inverse \(g^{(ff)} = g^{(46)} = \{62\}\). 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:

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

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:

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

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:

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

where:

\[
c_0 = a_0 \cdot b_0
\]

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

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

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

\[
c_4 = a_3 \cdot b_2 \oplus a_2 \cdot b_0 \oplus a_1 \cdot b_3
\]

\[
c_5 = a_3 \cdot b_2 \oplus a_2 \cdot b_1 \oplus a_1 \cdot b_2 \oplus a_0 \cdot b_3
\]

\[
c_6 = a_3 \cdot b_3
\]
Where \( \star \) 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:

\[
\begin{align*}
\hat{d}_3 &= a_3 \star b_0 \oplus a_2 \star b_1 \oplus a_1 \star b_2 \oplus a_0 \star b_3 \\
\hat{d}_2 &= a_2 \star b_0 \oplus a_1 \star b_1 \oplus a_0 \star b_2 \oplus a_3 \star b_0 \\
\hat{d}_1 &= a_1 \star b_0 \oplus a_0 \star b_1 \oplus a_3 \star b_2 \oplus a_2 \star b_3 \\
\hat{d}_0 &= a_0 \star b_0 \oplus a_3 \star b_1 \oplus a_2 \star b_2 \oplus a_1 \star b_3
\end{align*}
\]

(2.5.5)

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

\[
\begin{pmatrix}
\hat{d}_3 \\
\hat{d}_2 \\
\hat{d}_1 \\
\hat{d}_0
\end{pmatrix} =
\begin{pmatrix}
a_0 & a_1 & a_2 & a_3 \\
a_3 & a_0 & a_1 & a_2 \\
a_2 & a_3 & a_0 & a_1 \\
a_1 & a_2 & a_3 & a_0
\end{pmatrix}
\begin{pmatrix}
b_3 \\
b_2 \\
b_1 \\
b_0
\end{pmatrix}
\]

(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:

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

(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 round function in a number of cycles. The number of cycles \( Nn \) 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 \( Nc \) word sub-array 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, \( Nc \), 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, \( Nk \) 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 \( Nn + 1 \) individual round keys, each of which is itself an array of \( Nc \) 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
  state = in // the array k contains Nn + 1 individual round
            // 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.

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:

   \[ b'_i = b_i \oplus b_{(i+4) \mod 8} \oplus b_{(i+5) \mod 8} \oplus b_{(i+6) \mod 8} \oplus b_{(i+7) \mod 8} \oplus c_i \]  \hspace{1cm} (3.1.1)

   for 0 \leq 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:

Dr. Brian Gladman, v3.16, 11th August 2007
The final result of this two stage transformation is given in the following table.

<table>
<thead>
<tr>
<th>hex</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
<td>7c</td>
<td>77</td>
<td>7b</td>
<td>f2</td>
<td>6b</td>
<td>6f</td>
<td>c5</td>
<td>30</td>
<td>01</td>
<td>67</td>
<td>2b</td>
<td>fe</td>
<td>d7</td>
<td>ab</td>
<td>76</td>
</tr>
<tr>
<td>1</td>
<td>ca</td>
<td>82</td>
<td>c9</td>
<td>7d</td>
<td>fa</td>
<td>59</td>
<td>47</td>
<td>f0</td>
<td>ad</td>
<td>d4</td>
<td>a2</td>
<td>af</td>
<td>9c</td>
<td>a4</td>
<td>72</td>
<td>c0</td>
</tr>
<tr>
<td>2</td>
<td>b7</td>
<td>fd</td>
<td>93</td>
<td>26</td>
<td>36</td>
<td>3f</td>
<td>f7</td>
<td>cc</td>
<td>34</td>
<td>a5</td>
<td>e5</td>
<td>f1</td>
<td>71</td>
<td>d8</td>
<td>31</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>04</td>
<td>c7</td>
<td>23</td>
<td>c3</td>
<td>18</td>
<td>96</td>
<td>05</td>
<td>9a</td>
<td>07</td>
<td>12</td>
<td>80</td>
<td>e2</td>
<td>eb</td>
<td>27</td>
<td>b2</td>
<td>75</td>
</tr>
<tr>
<td>4</td>
<td>09</td>
<td>83</td>
<td>2c</td>
<td>1a</td>
<td>1b</td>
<td>6e</td>
<td>5a</td>
<td>a0</td>
<td>52</td>
<td>3b</td>
<td>d6</td>
<td>b3</td>
<td>29</td>
<td>e3</td>
<td>2f</td>
<td>84</td>
</tr>
<tr>
<td>5</td>
<td>53</td>
<td>d1</td>
<td>00</td>
<td>ed</td>
<td>20</td>
<td>fc</td>
<td>b1</td>
<td>5b</td>
<td>6a</td>
<td>cb</td>
<td>be</td>
<td>39</td>
<td>4a</td>
<td>4c</td>
<td>58</td>
<td>cf</td>
</tr>
<tr>
<td>6</td>
<td>df</td>
<td>ef</td>
<td>aa</td>
<td>fb</td>
<td>43</td>
<td>4d</td>
<td>33</td>
<td>85</td>
<td>45</td>
<td>f9</td>
<td>02</td>
<td>7f</td>
<td>50</td>
<td>3c</td>
<td>9f</td>
<td>a8</td>
</tr>
<tr>
<td>7</td>
<td>51</td>
<td>a3</td>
<td>40</td>
<td>8f</td>
<td>92</td>
<td>9d</td>
<td>36</td>
<td>f5</td>
<td>bc</td>
<td>b6</td>
<td>da</td>
<td>21</td>
<td>10</td>
<td>ff</td>
<td>f3</td>
<td>d2</td>
</tr>
<tr>
<td>8</td>
<td>cd</td>
<td>0c</td>
<td>13</td>
<td>ec</td>
<td>5f</td>
<td>97</td>
<td>44</td>
<td>17</td>
<td>c4</td>
<td>a7</td>
<td>7e</td>
<td>3d</td>
<td>64</td>
<td>5d</td>
<td>19</td>
<td>73</td>
</tr>
<tr>
<td>9</td>
<td>60</td>
<td>81</td>
<td>4f</td>
<td>dc</td>
<td>22</td>
<td>2a</td>
<td>90</td>
<td>46</td>
<td>ee</td>
<td>b8</td>
<td>14</td>
<td>de</td>
<td>5e</td>
<td>0b</td>
<td>db</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>e0</td>
<td>32</td>
<td>3a</td>
<td>0a</td>
<td>49</td>
<td>06</td>
<td>24</td>
<td>5c</td>
<td>c2</td>
<td>d3</td>
<td>ac</td>
<td>62</td>
<td>91</td>
<td>95</td>
<td>e4</td>
<td>79</td>
</tr>
<tr>
<td>b</td>
<td>e7</td>
<td>c8</td>
<td>37</td>
<td>6d</td>
<td>8d</td>
<td>d5</td>
<td>4e</td>
<td>a9</td>
<td>6c</td>
<td>56</td>
<td>f4</td>
<td>ea</td>
<td>65</td>
<td>7a</td>
<td>ae</td>
<td>08</td>
</tr>
<tr>
<td>c</td>
<td>ba</td>
<td>78</td>
<td>25</td>
<td>2e</td>
<td>1c</td>
<td>a6</td>
<td>b4</td>
<td>c6</td>
<td>e8</td>
<td>dd</td>
<td>74</td>
<td>4f</td>
<td>4b</td>
<td>bd</td>
<td>9b</td>
<td>8a</td>
</tr>
<tr>
<td>d</td>
<td>0e</td>
<td>b5</td>
<td>66</td>
<td>48</td>
<td>03</td>
<td>56</td>
<td>0e</td>
<td>61</td>
<td>53</td>
<td>5b</td>
<td>97</td>
<td>86</td>
<td>c1</td>
<td>1d</td>
<td>9e</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>e3</td>
<td>f9</td>
<td>98</td>
<td>11</td>
<td>69</td>
<td>d9</td>
<td>8e</td>
<td>94</td>
<td>9b</td>
<td>1e</td>
<td>87</td>
<td>e9</td>
<td>ce</td>
<td>55</td>
<td>28</td>
<td>df</td>
</tr>
<tr>
<td>f</td>
<td>8c</td>
<td>a1</td>
<td>89</td>
<td>0d</td>
<td>bf</td>
<td>e6</td>
<td>42</td>
<td>68</td>
<td>41</td>
<td>99</td>
<td>2d</td>
<td>0f</td>
<td>b0</td>
<td>54</td>
<td>bb</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 5 – The Substitution Table – Sbox[xy] (in hexadecimal)

The pseudo code for this transformation is as follows.

```plaintext
SubBytes(byte state[4,Nc], Nc)
begin
    for r = 0 step 1 to 3
        for c = 0 step 1 to Nc - 1
            state[r,c] = Sbox[state[r,c]]
        end for
    end for
end
```

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:

\[ s_{r,c} = s_{r,\{c+h[r,Nc]\}_4 \mod Nc} \quad \text{for} \quad 0 \leq r < 4 \quad \text{and} \quad 0 \leq c < Nc \]  

(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>( 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>

Note that the AES block size, \( Nc \), is fixed at 4

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.

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

**Figure 3** – *ShiftRows* acts independently on rows in the state

The pseudo code for this transformation is as follows.

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

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 2.5.

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.

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

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

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

**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
\[ state[r,c] = FFmul(0x02, t[r]) \oplus FFmul(0x03, t[(r + 1) \mod 4]) \oplus t[(r + 2) \mod 4] \oplus t[(r + 3) \mod 4] \]
end for
end

3.4 The XorRoundKey Transformation

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

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

(3.4.1)

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

![Diagram of words from the key schedule XOR'd into columns in the state](image)

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 xbyte(\( r, w \)) extracts byte \( r \) from word \( w \).

\begin{verbatim}
XorRoundKey(byte state[4,Nc], word k[\text{round,*}], Nc)
Begin
for c = 0 step 1 to Nc - 1
for r = 0 step 1 to 3
state[r,c] = state[r,c] \oplus xbyte(r, k[\text{round,c}])
end for
end
end
\end{verbatim}

4. The Key Schedule

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

The expansion of the input key into the key schedule proceeds according to the following pseudo code. The function 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 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 \([0, 0, u, x^{i-1}]\) with \(x^{i-1}\) being the powers of \(x\) in the field \(\text{GF}(256)\) discussed in section 2.3 (note that the index \(i\) starts at 1).

KeyExpansion(byte key[4*Nk], word k[Nn+1,Nc], Nc, NK, Nn)
begin
  i = 0
  while (i < NK)
    k[i] = word [key[4*i+3], key[4*i+2], key[4*i+1], key[4*i]]
    i = i + 1
  end while
  i = NK
  while (i < Nc + (NK + 1))
    word temp = k[i - 1]
    if (i mod NK = 0) or ((NK > 6) and (i mod NK = 4))
      temp = SubWord(temp)
    end if
    if (i mod NK = 0)
      temp = RotWord(temp) xor Rcon[i / NK]
    end if
    k[i] = k[i - NK] xor temp
    i = i + 1
  end while
end

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.

\[
\begin{array}{ccccccccccccccccccc}
  k_0 & k_1 & k_2 & k_3 & k_4 & k_5 & k_6 & k_7 & k_8 & k_9 & k_{10} & k_{11} & k_{12} & k_{13} & k_{14} & k_{15} & k_{16} & k_{17} & \ldots \\
\end{array}
\]

round key 0 k[0,*]  |  round key 1 k[1,*]  |  round key 2 k[2,*]  \ldots

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.

InvCipher(byte in[4*Nc], byte out[4*Nc], word k[Nn+1,Nc], Nc, Nn)
begin
  byte state[4,Nc]
  state = in
  XorRoundKey(state, k[Nn,-], Nc) // k[Nn*Nc..(Nn+1)*Nc-1]
  for round = Nn - 1 step -1 to 1
    InvShiftRows(state, Nc)
    InvSubBytes(state, Nc)
    XorRoundKey(state, k[round,-], Nc) // k[round*Nc..(round+1)*Nc-1]
    InvMixColumns(state, Nc)
  end for
  InvShiftRows(state, Nc)
  InvSubBytes(state, Nc)
  XorRoundKey(state, k[0,-], Nc) // k[0..Nc-1]
  out = state
end

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

$$s'_r[c+h(r,Nc)] \mod Nc = s_{r,c} \quad \text{for} \quad 0 \leq r < 4 \quad \text{and} \quad 0 \leq c < Nc$$  \hspace{1cm} (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.

\texttt{InvShiftRows(byte state[4,Nc], Nc)}

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

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:

\texttt{InvSubBytes(byte state[4,Nc], Nc)}

\begin{verbatim}
begin
  for r = 0 step 1 to 3
    for c = 0 step 1 to Nc - 1
      state[r,c] = InvSbox[state[r,c]]
    end for
  end for
end
\end{verbatim}

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

$$b'_i = b_{(i+2) \mod 8} \oplus b_{(i+5) \mod 8} \oplus b_{(i+7) \mod 8} \oplus d_i \quad \text{where byte} \quad d = \{05\} \hspace{1cm} (5.2.1)$$

<table>
<thead>
<tr>
<th>\texttt{hex}</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>52</td>
<td>09</td>
<td>6a</td>
<td>d5</td>
<td>30</td>
<td>36</td>
<td>a5</td>
<td>38</td>
<td>bf</td>
<td>40</td>
<td>a3</td>
<td>9e</td>
<td>81</td>
<td>f3</td>
</tr>
<tr>
<td>1</td>
<td>7c</td>
<td>e3</td>
<td>39</td>
<td>82</td>
<td>9b</td>
<td>2f</td>
<td>ff</td>
<td>87</td>
<td>34</td>
<td>8e</td>
<td>43</td>
<td>44</td>
<td>c4</td>
<td>de</td>
</tr>
<tr>
<td>2</td>
<td>54</td>
<td>7b</td>
<td>94</td>
<td>32</td>
<td>a6</td>
<td>c2</td>
<td>23</td>
<td>3d</td>
<td>ee</td>
<td>4c</td>
<td>95</td>
<td>0b</td>
<td>42</td>
<td>fa</td>
</tr>
<tr>
<td>3</td>
<td>08</td>
<td>2e</td>
<td>a1</td>
<td>66</td>
<td>28</td>
<td>d9</td>
<td>24</td>
<td>b2</td>
<td>76</td>
<td>5b</td>
<td>a2</td>
<td>49</td>
<td>6d</td>
<td>8b</td>
</tr>
<tr>
<td>4</td>
<td>72</td>
<td>f8</td>
<td>f6</td>
<td>64</td>
<td>86</td>
<td>68</td>
<td>98</td>
<td>16</td>
<td>d4</td>
<td>a4</td>
<td>5c</td>
<td>cc</td>
<td>5d</td>
<td>65</td>
</tr>
<tr>
<td>5</td>
<td>6c</td>
<td>70</td>
<td>48</td>
<td>50</td>
<td>fd</td>
<td>ed</td>
<td>b9</td>
<td>da</td>
<td>5e</td>
<td>15</td>
<td>46</td>
<td>5d</td>
<td>a7</td>
<td>8d</td>
</tr>
<tr>
<td>6</td>
<td>90</td>
<td>d8</td>
<td>ab</td>
<td>00</td>
<td>8c</td>
<td>bc</td>
<td>d3</td>
<td>0a</td>
<td>f7</td>
<td>e4</td>
<td>58</td>
<td>05</td>
<td>b8</td>
<td>6d</td>
</tr>
<tr>
<td>7</td>
<td>d0</td>
<td>2c</td>
<td>1e</td>
<td>8f</td>
<td>ca</td>
<td>3f</td>
<td>0e</td>
<td>02</td>
<td>c1</td>
<td>af</td>
<td>bd</td>
<td>03</td>
<td>01</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>3a</td>
<td>91</td>
<td>11</td>
<td>4f</td>
<td>67</td>
<td>dc</td>
<td>ea</td>
<td>97</td>
<td>f2</td>
<td>cf</td>
<td>ce</td>
<td>f0</td>
<td>b4</td>
<td>e6</td>
</tr>
<tr>
<td>9</td>
<td>96</td>
<td>ac</td>
<td>74</td>
<td>22</td>
<td>e7</td>
<td>ad</td>
<td>35</td>
<td>85</td>
<td>e5</td>
<td>f9</td>
<td>37</td>
<td>e8</td>
<td>0c</td>
<td>75</td>
</tr>
<tr>
<td>a</td>
<td>47</td>
<td>f1</td>
<td>1a</td>
<td>71</td>
<td>1d</td>
<td>29</td>
<td>c5</td>
<td>89</td>
<td>6f</td>
<td>b7</td>
<td>62</td>
<td>0e</td>
<td>aa</td>
<td>18</td>
</tr>
<tr>
<td>b</td>
<td>fc</td>
<td>56</td>
<td>3e</td>
<td>4b</td>
<td>c5</td>
<td>d2</td>
<td>79</td>
<td>20</td>
<td>9a</td>
<td>db</td>
<td>c0</td>
<td>fe</td>
<td>78</td>
<td>cd</td>
</tr>
<tr>
<td>c</td>
<td>1f</td>
<td>dd</td>
<td>a8</td>
<td>33</td>
<td>88</td>
<td>07</td>
<td>c7</td>
<td>31</td>
<td>8b</td>
<td>12</td>
<td>10</td>
<td>59</td>
<td>27</td>
<td>80</td>
</tr>
<tr>
<td>d</td>
<td>60</td>
<td>51</td>
<td>7f</td>
<td>a9</td>
<td>19</td>
<td>b5</td>
<td>4a</td>
<td>0d</td>
<td>2d</td>
<td>ec</td>
<td>57</td>
<td>a5</td>
<td>9f</td>
<td>39</td>
</tr>
<tr>
<td>e</td>
<td>a0</td>
<td>e0</td>
<td>3b</td>
<td>4d</td>
<td>ae</td>
<td>2a</td>
<td>f5</td>
<td>b0</td>
<td>c8</td>
<td>eb</td>
<td>bb</td>
<td>3c</td>
<td>83</td>
<td>53</td>
</tr>
<tr>
<td>f</td>
<td>17</td>
<td>2b</td>
<td>04</td>
<td>7e</td>
<td>ba</td>
<td>77</td>
<td>d6</td>
<td>2a</td>
<td>26</td>
<td>e1</td>
<td>69</td>
<td>14</td>
<td>63</td>
<td>55</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>y</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>52</td>
<td>09</td>
<td>6a</td>
<td>d5</td>
<td>30</td>
<td>36</td>
<td>a5</td>
<td>38</td>
<td>bf</td>
<td>40</td>
<td>a3</td>
<td>9e</td>
<td>81</td>
<td>f3</td>
</tr>
<tr>
<td>1</td>
<td>7c</td>
<td>e3</td>
<td>39</td>
<td>82</td>
<td>9b</td>
<td>2f</td>
<td>ff</td>
<td>87</td>
<td>34</td>
<td>8e</td>
<td>43</td>
<td>44</td>
<td>c4</td>
<td>de</td>
</tr>
<tr>
<td>2</td>
<td>54</td>
<td>7b</td>
<td>94</td>
<td>32</td>
<td>a6</td>
<td>c2</td>
<td>23</td>
<td>3d</td>
<td>ee</td>
<td>4c</td>
<td>95</td>
<td>0b</td>
<td>42</td>
<td>fa</td>
</tr>
<tr>
<td>3</td>
<td>08</td>
<td>2e</td>
<td>a1</td>
<td>66</td>
<td>28</td>
<td>d9</td>
<td>24</td>
<td>b2</td>
<td>76</td>
<td>5b</td>
<td>a2</td>
<td>49</td>
<td>6d</td>
<td>8b</td>
</tr>
<tr>
<td>4</td>
<td>72</td>
<td>f8</td>
<td>f6</td>
<td>64</td>
<td>86</td>
<td>68</td>
<td>98</td>
<td>16</td>
<td>d4</td>
<td>a4</td>
<td>5c</td>
<td>cc</td>
<td>5d</td>
<td>b6</td>
</tr>
<tr>
<td>5</td>
<td>6c</td>
<td>70</td>
<td>48</td>
<td>50</td>
<td>fd</td>
<td>ed</td>
<td>b9</td>
<td>da</td>
<td>5e</td>
<td>15</td>
<td>46</td>
<td>5d</td>
<td>a7</td>
<td>9d</td>
</tr>
<tr>
<td>6</td>
<td>90</td>
<td>d8</td>
<td>ab</td>
<td>00</td>
<td>8c</td>
<td>bc</td>
<td>d3</td>
<td>0a</td>
<td>f7</td>
<td>e4</td>
<td>58</td>
<td>05</td>
<td>b8</td>
<td>6d</td>
</tr>
<tr>
<td>7</td>
<td>d0</td>
<td>2c</td>
<td>1e</td>
<td>8f</td>
<td>ca</td>
<td>3f</td>
<td>0e</td>
<td>02</td>
<td>c1</td>
<td>af</td>
<td>bd</td>
<td>03</td>
<td>01</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>3a</td>
<td>91</td>
<td>11</td>
<td>4f</td>
<td>67</td>
<td>dc</td>
<td>ea</td>
<td>97</td>
<td>f2</td>
<td>cf</td>
<td>ce</td>
<td>f0</td>
<td>b4</td>
<td>e6</td>
</tr>
<tr>
<td>9</td>
<td>96</td>
<td>ac</td>
<td>74</td>
<td>22</td>
<td>e7</td>
<td>ad</td>
<td>35</td>
<td>85</td>
<td>e5</td>
<td>f9</td>
<td>37</td>
<td>e8</td>
<td>0c</td>
<td>75</td>
</tr>
<tr>
<td>a</td>
<td>47</td>
<td>f1</td>
<td>1a</td>
<td>71</td>
<td>1d</td>
<td>29</td>
<td>c5</td>
<td>89</td>
<td>6f</td>
<td>b7</td>
<td>62</td>
<td>0e</td>
<td>aa</td>
<td>18</td>
</tr>
<tr>
<td>b</td>
<td>fc</td>
<td>56</td>
<td>3e</td>
<td>4b</td>
<td>c5</td>
<td>d2</td>
<td>79</td>
<td>20</td>
<td>9a</td>
<td>db</td>
<td>c0</td>
<td>fe</td>
<td>78</td>
<td>cd</td>
</tr>
<tr>
<td>c</td>
<td>1f</td>
<td>dd</td>
<td>a8</td>
<td>33</td>
<td>88</td>
<td>07</td>
<td>c7</td>
<td>31</td>
<td>8b</td>
<td>12</td>
<td>10</td>
<td>59</td>
<td>27</td>
<td>80</td>
</tr>
<tr>
<td>d</td>
<td>60</td>
<td>51</td>
<td>7f</td>
<td>a9</td>
<td>19</td>
<td>b5</td>
<td>4a</td>
<td>0d</td>
<td>2d</td>
<td>ec</td>
<td>57</td>
<td>a5</td>
<td>9f</td>
<td>39</td>
</tr>
<tr>
<td>e</td>
<td>a0</td>
<td>e0</td>
<td>3b</td>
<td>4d</td>
<td>ae</td>
<td>2a</td>
<td>f5</td>
<td>b0</td>
<td>c8</td>
<td>eb</td>
<td>bb</td>
<td>3c</td>
<td>83</td>
<td>53</td>
</tr>
<tr>
<td>f</td>
<td>17</td>
<td>2b</td>
<td>04</td>
<td>7e</td>
<td>ba</td>
<td>77</td>
<td>d6</td>
<td>2a</td>
<td>26</td>
<td>e1</td>
<td>69</td>
<td>14</td>
<td>63</td>
<td>55</td>
</tr>
</tbody>
</table>

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.

\[
\begin{align*}
S_{3,c} &= \begin{bmatrix} 0e & 09 & 0d & 0b \\ 0b & 0e & 09 & 0d \\ 0d & 0b & 0e & 09 \\ 09 & 0d & 0b & 0e \end{bmatrix} \quad \text{for} \quad 0 \leq c < Nc \\
S_{2,c} &= \begin{bmatrix} 0e & 09 & 0d & 0b \\ 0b & 0e & 09 & 0d \\ 0d & 0b & 0e & 09 \\ 09 & 0d & 0b & 0e \end{bmatrix} \\
S_{1,c} &= \begin{bmatrix} 0e & 09 & 0d & 0b \\ 0b & 0e & 09 & 0d \\ 0d & 0b & 0e & 09 \\ 09 & 0d & 0b & 0e \end{bmatrix} \\
S_{0,c} &= \begin{bmatrix} 0e & 09 & 0d & 0b \\ 0b & 0e & 09 & 0d \\ 0d & 0b & 0e & 09 \\ 09 & 0d & 0b & 0e \end{bmatrix}
\end{align*}
\]

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.

\[
\text{InvMixColumns}(\text{byte block}[4, Nc], Nc) \\
\text{begin} \\
\quad \text{byte } t[4] \\
\quad \text{for } c = 0 \text{ step } 1 \text{ to } Nc - 1 \\
\quad \quad \text{for } r = 0 \text{ step } 1 \text{ to } 3 \\
\quad \quad \quad t[r] = \text{block}[r,c] \\
\quad \quad \text{end for} \\
\quad \text{for } r = 0 \text{ step } 1 \text{ to } 3 \\
\quad \quad \text{block}[r,c] = \\
\quad \quad \quad \text{FFmul}(0x00a, t[r]) \text{ xor} \\
\quad \quad \quad \text{FFmul}(0x00b, t[(r + 1) \mod 4]) \text{ xor} \\
\quad \quad \quad \text{FFmul}(0x00d, t[(r + 2) \mod 4]) \text{ xor} \\
\quad \quad \quad \text{FFmul}(0x009, t[(r + 3) \mod 4]) \\
\quad \text{end for} \\
\text{end for} \\
\text{end}
\]

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:

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

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.

\[Dr. Brian Gladman, v3.16, 1st August 2007\]
In this modified form the inverse cipher is as follows (with the modified decryption key schedule in the word array \(dk[Nn+1,Nc]\)).

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

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).

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

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 have 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 $0x6$. 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 (0x1b) is
  endif // not needed here since bb is an 8 bit value
  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 array 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:

\[ c[3] = \{02\} \cdot c[3] \oplus \{03\} \cdot c[0] \oplus c[1] \oplus c[2] \]
\[ c[2] = \{02\} \cdot c[2] \oplus \{03\} \cdot c[3] \oplus c[0] \oplus c[1] \]
\[ c[1] = \{02\} \cdot c[1] \oplus \{03\} \cdot c[2] \oplus c[3] \oplus c[0] \] (7.2.1)
\[ c[0] = \{02\} \cdot c[0] \oplus \{03\} \cdot c[1] \oplus c[2] \oplus c[3] \]

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:
\[ c[3]' = v \odot c[3] \oplus \{02\} \odot (c[3] \oplus c[0]) \\
\]
\[ c[2]' = v \odot c[2] \oplus \{02\} \odot (c[2] \oplus c[3]) \\
\]
\[ c[1]' = v \odot c[1] \oplus \{02\} \odot (c[1] \oplus c[2]) \\
\]
\[ c[0]' = v \odot c[0] \oplus \{02\} \odot (c[0] \oplus c[1]) \]

(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 \( v \)):

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

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 2, 0 and 1 respectively and moves the byte in position 3 to position 0.

- \( \text{rot2}(w) \) moves the bytes in positions 0, 1 and 3 in \( w \) to positions 2, 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:

```plaintext
word FFmulX(const word w)
begin
word t = w & 0x80808080
return ((w ^ t) << 1) ^ ((t >> 3) | (t >> 4) | (t >> 6) | (t >> 7))
end
```

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 \( (t >> 3) | (t >> 4) | (t >> 6) | (t >> 7) \) 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 \( ((u - (u >> 7)) \land 0x1b1b1b1b1b) \) or \( (u >> 7) \land 0x000000001b \), 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:

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

(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:

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

(7.3.2)

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

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

(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:

\[
\begin{align*}
u &= c[1] \wedge c[0] \\
v &= t \wedge u \\
w &= v \wedge FFmul(0x08, v) \\
c[3]' &= \{04\} \cdot (c[3] \oplus c[1]) \oplus (v \oplus c[3] \oplus c[0]) \cdot (c[3] \oplus c[0]) \\
c[2]' &= \{04\} \cdot (c[2] \oplus c[0]) \oplus (v \oplus c[2] \oplus c[0]) \cdot (c[2] \oplus c[3]) \\
c[1]' &= \{04\} \cdot (c[1] \oplus c[3]) \oplus (v \oplus c[1] \oplus c[0]) \cdot (c[1] \oplus c[2]) \\
c[0]' &= \{04\} \cdot (c[0] \oplus c[2]) \oplus (v \oplus c[0] \oplus c[2]) \cdot (c[0] \oplus c[1])
\end{align*}
\]

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 SubBytes, ShiftRows, MixColumns and XorRoundKey transformations in turn gives:

After SubBytes:
After ShiftRows:

\[
\begin{bmatrix}
S_{3,c} \\
S_{2,c} \\
S_{1,c} \\
S_{0,c}
\end{bmatrix}' =
\begin{bmatrix}
S_{[S_{3,c}]} \\
S_{[S_{2,c}]} \\
S_{[S_{1,c}]} \\
S_{[S_{0,c}]}
\end{bmatrix}
\]  \hspace{1cm} (7.4.1)

After MixColumns:

\[
\begin{bmatrix}
S_{3,c} \\
S_{2,c} \\
S_{1,c} \\
S_{0,c}
\end{bmatrix}'' =
\begin{bmatrix}
(S_{3,c}^{c+h(3,Nc)mod Nc}) \\
(S_{2,c}^{c+h(2,Nc)mod Nc}) \\
(S_{1,c}^{c+h(1,Nc)mod Nc}) \\
S_{0,c}
\end{bmatrix} =
\begin{bmatrix}
S_{[S_{3,c}]} \\
S_{[S_{2,c}]} \\
S_{[S_{1,c}]} \\
S_{[S_{0,c}]}
\end{bmatrix}
\]  \hspace{1cm} (7.4.2)

After XorRoundKey:

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

(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:

\[
\begin{bmatrix}
S_{3,c} \\
S_{2,c} \\
S_{1,c} \\
S_{0,c}
\end{bmatrix}' =
\begin{bmatrix}
[02] & [01] & [01] & [03] \\
[03] & [02] & [01] & [01] \\
[01] & [03] & [02] & [01] \\
[01] & [01] & [03] & [02]
\end{bmatrix}
\begin{bmatrix}
S_{[S_{3,c}]} \\
S_{[S_{2,c}]} \\
S_{[S_{1,c}]} \\
S_{[S_{0,c}]}
\end{bmatrix}
\oplus
\begin{bmatrix}
k_{3,c} \\
k_{2,c} \\
k_{1,c} \\
k_{0,c}
\end{bmatrix}
\]  \hspace{1cm} (7.4.5)

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

\[
T_{3}[x] =
\begin{bmatrix}
[02] & S[x] \\
[03] & S[x] \\
S[x] & S[x] \\
S[x] & S[x]
\end{bmatrix}
\]  \hspace{1cm} (7.4.6)

(7.4.6)

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

\[
\begin{bmatrix}
S_{3,c} \\
S_{2,c} \\
S_{1,c} \\
S_{0,c}
\end{bmatrix}' =
T_{3}[S_{3,c}(3)] \oplus T_{2}[S_{2,c}(2)] \oplus T_{1}[S_{1,c}(1)] \oplus T_{0}[S_{0,c}(0)] \oplus k_{round,c}
\]  \hspace{1cm} (7.4.7)

where \( c(r) = [c + h(r,Nc)] mod Nc \), \( c(0) = c \) and \( k_{round,c} \) 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:

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

(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 \(T_t[x] = \text{rot1}(T_{t-1}[x])\), 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:

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

(7.4.9)

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

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

(7.4.10)

where \(c(r) = [c - h(r, Nc)]mod Nc\), \(c(0) = c\) and \(k_{round,c}\) is word \(c\) of round key \(round\) 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 following diagram shows the hexadecimal values in the state array as the cipher progresses for a cipher input length (Nb) of 4 and a cipher key length (Nk) 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>
<th>Round</th>
<th>Start of round</th>
<th>After</th>
<th>After</th>
<th>After</th>
<th>After</th>
<th>Round</th>
<th>Key value</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 28 31 09</td>
<td>7e ae f7 cc</td>
<td>1</td>
<td>e6 02 08 36</td>
<td>6b 00 00 00</td>
<td>00 00 00 00</td>
<td>00 2b 0b 09</td>
<td>32 28 31 09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6c 3a 37 37</td>
<td>15 25</td>
<td>2</td>
<td>b0 3d 84 06</td>
<td>00 00 00 00</td>
<td>00 00 00 00</td>
<td>00 49 44 4d</td>
<td>6c 3a 37 37</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a8 8d a2 34</td>
<td>17 1a 5b 30</td>
<td>3</td>
<td>6a 9a 09 9c</td>
<td>00 00 00 00</td>
<td>00 00 00 00</td>
<td>00 52 52 52</td>
<td>a8 8d a2 34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19 a0 9a e9</td>
<td>3d 44</td>
<td>4</td>
<td>6e 86 06 9c</td>
<td>00 00 00 00</td>
<td>00 00 00 00</td>
<td>00 00 00 00</td>
<td>19 a0 9a e9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3d 44 09 06</td>
<td>3d 44</td>
<td>5</td>
<td>6e 86 06 9c</td>
<td>00 00 00 00</td>
<td>00 00 00 00</td>
<td>00 00 00 00</td>
<td>3d 44 09 06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6b 3a 0d 0a</td>
<td>3d 44</td>
<td>6</td>
<td>6e 86 06 9c</td>
<td>00 00 00 00</td>
<td>00 00 00 00</td>
<td>00 00 00 00</td>
<td>6b 3a 0d 0a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32 28 31 09</td>
<td>7e ae f7 cc</td>
<td>7</td>
<td>6b 00 00 00</td>
<td>00 00 00 00</td>
<td>00 00 00 00</td>
<td>00 2b 0b 09</td>
<td>32 28 31 09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6c 3a 37 37</td>
<td>15 25</td>
<td>8</td>
<td>b0 3d 84 06</td>
<td>00 00 00 00</td>
<td>00 00 00 00</td>
<td>00 49 44 4d</td>
<td>6c 3a 37 37</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a8 8d a2 34</td>
<td>17 1a 5b 30</td>
<td>9</td>
<td>6a 9a 09 9c</td>
<td>00 00 00 00</td>
<td>00 00 00 00</td>
<td>00 52 52 52</td>
<td>a8 8d a2 34</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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