THE ADVANCED ENCRYPTION STANDARD

William Stallings

c/o Cryptologia, Department of Mathematical Sciences, United States Military Academy, West Point NY 10996 USA. ws@shore.net.

To cite this article: William Stallings (2002): THE ADVANCED ENCRYPTION STANDARD, Cryptologia, 26:3, 165-188

To link to this article: http://dx.doi.org/10.1080/0161-110291890876
THE ADVANCED
ENCRYPTION STANDARD

William Stallings

ADDRESS: c/o Cryptologia, Department of Mathematical Sciences, United States Military Academy, West Point NY 10996 USA. ws@shore.net.

ABSTRACT: In this paper, we describe the Advanced Encryption Standard (AES), which has been approved after an international competition by the National Institute of Standards and Technology.

KEYWORDS: Advanced Encryption Standard, block cipher, symmetric cipher.

1 INTRODUCTION

The most widely used encryption scheme is based on the Data Encryption Standard (DES) adopted in 1977 by the National Bureau of Standards, now the National Institute of Standards and Technology (NIST), as Federal Information Processing Standard 46 (FIPS PUB 46). For DES, data are encrypted in 64-bit blocks using a 56-bit key. The algorithm transforms a 64-bit input in a series of steps into a 64-bit output. The same steps, with the same key, are used to reverse the encryption.

From the beginning, there were concerns about the vulnerability of DES because of its short key length. DES finally and definitively proved insecure in July 1998, when the Electronic Frontier Foundation (EFF) announced that it had broken a DES encryption using a special-purpose “DES cracker” machine that was built for less than $250,000. The attack took less than three days. The EFF has published a detailed description of the machine, enabling others to build their own cracker [1]. And, of course, hardware prices will continue to drop as speeds increase, making DES virtually worthless.

Whatever the merits of the case, DES has flourished and is widely used, especially in financial applications. In 1994, NIST reaffirmed DES for federal use for another five years; NIST recommended the use of DES for applications other than the protection of classified information. In 1999, NIST issued a new version of its standard (FIPS PUB 46-3) that indicated that DES should only be used for legacy systems and that triple DES (which in essence involves repeating the
DES algorithm three times on the plaintext using two or three different keys to produce the ciphertext), known as 3DES, be used.

3DES has two attractions that assure its widespread use over the next few years. First, with its 168-bit key length, it overcomes the vulnerability to brute-force attack of DES. Second, the underlying encryption algorithm in 3DES is the same as in DES. This algorithm has been subjected to more scrutiny than any other encryption algorithm over a longer period of time, and no effective cryptanalytic attack based on the algorithm rather than brute force has been found. Accordingly, there is a high level of confidence that 3DES is very resistant to cryptanalysis. If security were the only consideration, then 3DES would be an appropriate choice for a standardized encryption algorithm for decades to come.

The principal drawback of 3DES is that the algorithm is relatively sluggish in software. The original DES was designed for mid-1970s hardware implementation and does not produce efficient software code. 3DES, which has three times as many rounds as DES, is correspondingly slower. A secondary drawback is that both DES and 3DES use a 64-bit block size. For reasons of both efficiency and security, a larger block size is desirable.

Because of these drawbacks, 3DES is not a reasonable candidate for long-term use. As a replacement, NIST in 1997 issued a call for proposals for a new Advanced Encryption Standard (AES), which should have a security strength equal to or better than 3DES and significantly improved efficiency. In addition to these general requirements, NIST specified that AES must be a symmetric block cipher with a block length of 128 bits and support for key lengths of 128, 192, and 256 bits.

In a first round of evaluation, 15 proposed algorithms were accepted. A second round narrowed the field to 5 algorithms. NIST completed its evaluation process and published a final standard (FIPS PUB 197) in November of 2001. NIST selected Rijndael as the proposed AES algorithm [3, 4, 6]. The two researchers who developed and submitted Rijndael for the AES are both cryptographers from Belgium: Dr. Joan Daemen and Dr. Vincent Rijmen.

Ultimately, AES is intended to replace 3DES, but this process will take a number of years. NIST anticipates that 3DES will remain an approved algorithm (for U. S. Government use) for the foreseeable future.

2 THE AES CIPHER

The Rijndael proposal for AES defined a cipher in which the block length and the key length can be independently specified to be 128, 192, or 256 bits. The AES specification uses the same three key size alternatives but limits the block length
Figure 1. AES Encryption and Decryption.
to 128 bits. A number of AES parameters depend on the key length (Table 1). In the description of this paper, we assume a key length of 128 bits, which is likely to be the one most commonly implemented.

<table>
<thead>
<tr>
<th>Key size (words/bytes/bits)</th>
<th>4/16/128</th>
<th>6/24/192</th>
<th>8/32/256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plaintext block size (words/bytes/bits)</td>
<td>4/16/128</td>
<td>4/16/128</td>
<td>4/16/128</td>
</tr>
<tr>
<td>Numbers of rounds</td>
<td>10</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>Round keysize (words/bytes/bits)</td>
<td>4/16/128</td>
<td>4/16/128</td>
<td>4/16/128</td>
</tr>
<tr>
<td>Expanded key size (words/bytes/bits)</td>
<td>44/176</td>
<td>52/208</td>
<td>60/240</td>
</tr>
</tbody>
</table>

Table 1. AES parameters.

Rijndael was designed to have the following characteristics:

- Resistance against all known attacks
- Speed and code compactness on a wide range of platforms
- Design simplicity

Figure 1 shows the overall structure of AES. The input to the encryption and decryption algorithms is a single 128-bit block. In FIPS PUB 197, this block is depicted as a square matrix of bytes. This block is copied into the State array, which is modified at each stage of encryption or decryption. After the final stage, State is copied to an output matrix. These operations are depicted in Figure 2a. Similarly, the 128-bit key is depicted as a square matrix of bytes. This key is then expanded into an array of key schedule words; each word is four bytes and the total key schedule is 44 words for the 128-bit key (Figure 2b). Note that the ordering of bytes within a matrix is by column. So, for example, the first four bytes of a 128-bit plaintext input to the encryption cipher occupy the first column of the in matrix, the second four bytes occupy the second column, and so on. Similarly, the first four bytes of the expanded key, which form a word, occupy the first column of the w matrix.

Before delving into details, we can make several comments about the overall AES structure:

1. AES does not have a Feistel structure, unlike DES and many other ciphers. In the classic Feistel structure, half of the data block is used to modify the other half of the data block, and then the halves are swapped. Two of the AES finalists, including Rijndael, do not use a Feistel structure but process the entire data block in parallel during each round using substitutions and permutation.
2. The key that is provided as input is expanded into an array of forty-four 32-bit words, \( w[i] \). Four distinct words (128 bits) serve as a round key for each round; these are indicated in Figure 1.

3. Four different stages are used, one of permutation and three of substitution:
   - **Substitute bytes:** Uses an S-box to perform a byte-by-byte substitution of the block
   - **Shift rows:** A simple permutation
   - **Mix columns:** A substitution that makes use of arithmetic over \( GF(2^8) \)
   - **Add round key:** A simple bitwise XOR of the current block with a portion of the expanded key

4. The structure is quite simple. For both encryption and decryption, the cipher begins with an Add Round Key stage, followed by 9 rounds that each include all four stages, followed by a tenth round of three stages.

5. Only the Add Round Key stage makes use of the key. For this reason, the cipher begins and ends with an Add Round Key stage. Any other stage, applied at the beginning or end, is reversible without knowledge of the key and so would add no security.
6. The Add Round Key stage is, in effect, a form of Vernam cipher and by itself would not be formidable. The other three stages together provide confusion, diffusion, and nonlinearity, but by themselves would provide no security because they do not use the key. We can view the cipher as alternating operations of XOR encryption (Add Round Key) of a block, followed by scrambling of the block (the other three stages), followed by XOR encryption, and so on. This scheme is both efficient and highly secure.

7. Each stage is easily reversible. For the Substitute Byte, Shift Row, and Mix Columns stages, an inverse function is used in the decryption algorithm. For the Add Round Key stage, the inverse is achieved by XORing the same round key to the block, using the result that $A \oplus A \oplus B = B$.

8. As with most block ciphers, the decryption algorithm makes use of the expanded key in reverse order. However, the decryption algorithm is not identical to the encryption algorithm. This is a consequence of the particular structure of AES.

9. Once it is established that all four stages are reversible, it is easy to verify that decryption does recover the plaintext. Figure 1 lays out encryption and decryption going in opposite vertical directions. At each horizontal point (e.g., the dashed line in the figure), State is the same for both encryption and decryption.

10. The final round of both encryption and decryption consists of only three stages. Again, this is a consequence of the particular structure of AES and is required to make the cipher reversible.

AES uses arithmetic in the finite field $GF(2^8)$, with the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$. The developers of Rijndael give as their motivation for selecting this one of the 30 possible irreducible polynomials of degree 8 that it is the first one on the list given in [5]. The Appendix provides a brief introduction to finite fields.

**Substitute Bytes Transformation**

The forward substitute byte transformation, called SubBytes, is a simple table lookup. AES defines a $16 \times 16$ matrix of byte values, called an S-box (Table 2a), that contains a permutation of all possible 256 8-bit values. Each individual byte of State is mapped into a new byte in the following way: The leftmost four bits of the byte are used as a row value and the rightmost four bits are used as a column value. These row and column values serve as indexes into the S-box to select a

---

1In the remainder of this discussion, references to $GF(2^8)$ refer to the finite field defined with this polynomial.
unique 8-bit output value. For example, the hexadecimal value 95 references row 9, column 5 of the S-box, which contains the value 2A. Accordingly, the value 95 is mapped into the value 2A.

Here is an example of the SubBytes transformation:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>EA</td>
<td>04</td>
<td>65</td>
<td>85</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>45</td>
<td>5D</td>
<td>96</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5C</td>
<td>33</td>
<td>98</td>
<td>B0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F0</td>
<td>2D</td>
<td>AD</td>
<td>C5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \Rightarrow \]

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>87</td>
<td>F2</td>
<td>4D</td>
<td>97</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EC</td>
<td>6E</td>
<td>4C</td>
<td>90</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4A</td>
<td>C3</td>
<td>46</td>
<td>E7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8C</td>
<td>D8</td>
<td>95</td>
<td>A6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The S-box is constructed in the following fashion.

1. Initialize the S-box with the byte values in ascending sequence row by row. The first row contains \{00\}, \{01\}, \{02\}, etc., the second row contains \{10\}, \{11\}, etc., and so on. Thus, the value of the byte at row \(x\), column \(y\) is \(xy\).

2. Map each byte in the S-box to its multiplicative inverse in the finite field \(GF(2^8)\); the value \{00\} is mapped to itself.

3. Consider that each byte in the S-box consists of 8 bits labeled \((b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0)\).

Apply the following transformation to each bit of each byte in the S-box:

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

where \(c_i\) is the \(i\)th bit of byte \(c\) with the value \{63\}, or 01100011 in binary. The prime (') indicates that the variable is to be updated by the value on the right. The AES standard depicts this transformation in matrix form as follows:

\[
\begin{bmatrix}
    b'_0 \\
    b'_1 \\
    b'_2 \\
    b'_3 \\
    b'_4 \\
    b'_5 \\
    b'_6 \\
    b'_7
\end{bmatrix}
= \begin{bmatrix}
    1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
    1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
    1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
    1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
    1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
    0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
    0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
    0 & 0 & 0 & 1 & 1 & 1 & 1 & 1
\end{bmatrix}
\begin{bmatrix}
    b_0 \\
    b_1 \\
    b_2 \\
    b_3 \\
    b_4 \\
    b_5 \\
    b_6 \\
    b_7
\end{bmatrix}
+ \begin{bmatrix}
    1 \\
    1 \\
    0 \\
    0 \\
    0 \\
    0 \\
    0 \\
    0
\end{bmatrix}
\quad (2)
\]

In FIPS PUB 197, a hexadecimal number is indicated by enclosing it in curly brackets. We use that convention.

171
Equation (2) has to be interpreted carefully. In ordinary matrix multiplication, each element in the product matrix is the sum of products of the elements of one row and one column. In this case, each element in the product matrix is the bitwise XOR of products of elements of one row and one column. Further, the final addition shown in Equation (2) is a bitwise XOR.

As an example consider the input value \{95\}. The multiplicative inverse in \(GF(2^8)\) is \(\{95\}^{-1} = \{8A\}\), which is 10001010 in binary. Using Equation (2):

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
\end{bmatrix}
\begin{bmatrix}
0 \\
1 \\
0 \\
1 \\
0 \\
0 \\
0 \\
\end{bmatrix}
\oplus
\begin{bmatrix}
1 \\
1 \\
0 \\
1 \\
0 \\
0 \\
0 \\
\end{bmatrix}
\oplus
\begin{bmatrix}
1 \\
0 \\
0 \\
1 \\
0 \\
0 \\
0 \\
\end{bmatrix}
= \begin{bmatrix}
0 \\
1 \\
0 \\
0 \\
1 \\
1 \\
1 \\
0 \\
\end{bmatrix}
\]

The result is \{2A\}, which should appear in row \{09\} column 05 of the S-box. This is verified by checking Table 2a.

The inverse substitute byte transformation, called InvSubBytes, makes use of the inverse S-box shown in Table 2b. Note, for example, that the input \{2A\} produces the output \{95\}, and the input \{95\} to the S-box produces \{2A\}. The inverse S-box is constructed by applying the inverse of the transformation in Equation (1) followed by taking the multiplicative inverse in \(GF(2^8)\).

The S-box is designed to be resistant to known cryptanalytic attacks. Specifically, the Rijndael developers sought a design that has a low correlation between input bits and output bits, and the property that the output cannot be described as a simple mathematical function of the input [3]. In addition, the constant in Equation (1) was chosen so that the S-box has no fixed points \(S-box(a) = a\) and no "opposite fixed points" \(S-box(a) = \bar{a}\), where \(\bar{a}\) is the bitwise complement of \(a\).

Of course, the S-box must be invertible, that is, IS-box[\(S-box(a)\)] = \(a\). However, the S-box is not self-inverse in the sense that it is not true that S-box\(a\) = IS-box\(a\). For example, S-box (\{95\}) = \{2A\}, but IS-box\{95\}) = \{AD\}.

**Shift Row Transformation**

The **forward shift row transformation**, called ShiftRows, is depicted in (Figure 3a). The first row of State is not altered. For the second row, a 1-byte circular left shift is performed. For the third row, a 2-byte circular left shift is performed.
### Table 2a. S-box.

<table>
<thead>
<tr>
<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>FB</td>
<td>6B</td>
<td>6F</td>
<td>C5F2</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>
</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>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>0A</td>
<td>07</td>
<td>12</td>
<td>80</td>
<td>E2</td>
<td>EB</td>
<td>27</td>
<td>B2</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>
</tr>
<tr>
<td>5</td>
<td>55</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>
</tr>
<tr>
<td>6</td>
<td>D0</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>
</tr>
<tr>
<td>7</td>
<td>51</td>
<td>A5</td>
<td>40</td>
<td>8F</td>
<td>92</td>
<td>9D</td>
<td>38</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>
</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>
</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>88</td>
<td>46</td>
<td>EE</td>
<td>B3</td>
<td>14</td>
<td>DE</td>
<td>5E</td>
<td>0B</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>
</tr>
<tr>
<td>B</td>
<td>E7</td>
<td>C8</td>
<td>37</td>
<td>6D</td>
<td>8D</td>
<td>DS</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>
</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>1F</td>
<td>4B</td>
<td>BD</td>
<td>8B</td>
</tr>
<tr>
<td>D</td>
<td>70</td>
<td>3E</td>
<td>B5</td>
<td>66</td>
<td>48</td>
<td>03</td>
<td>F6</td>
<td>0E</td>
<td>61</td>
<td>35</td>
<td>57</td>
<td>B9</td>
<td>86</td>
<td>C1</td>
<td>1D</td>
</tr>
<tr>
<td>E</td>
<td>EI</td>
<td>F8</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>
</tr>
<tr>
<td>F</td>
<td>8C</td>
<td>Al</td>
<td>89</td>
<td>0D</td>
<td>BF</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 2b. Inverse S-box.

<table>
<thead>
<tr>
<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>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>
<td>D7</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>SE</td>
<td>43</td>
<td>44</td>
<td>C4</td>
<td>DE</td>
<td>E9</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>F2</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>
<td>D1</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>
<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>57</td>
<td>A7</td>
<td>8D</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>B3</td>
<td>45</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>0F</td>
<td>02</td>
<td>C1</td>
<td>AF</td>
<td>BD</td>
<td>03</td>
<td>01</td>
<td>13</td>
<td>8A</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>F6</td>
<td>73</td>
</tr>
<tr>
<td>9</td>
<td>96</td>
<td>AC</td>
<td>74</td>
<td>22</td>
<td>E7</td>
<td>35</td>
<td>85</td>
<td>E2</td>
<td>F9</td>
<td>37</td>
<td>E8</td>
<td>1C</td>
<td>75</td>
<td>DF</td>
<td>6E</td>
</tr>
<tr>
<td>A</td>
<td>47</td>
<td>F1</td>
<td>1A</td>
<td>71</td>
<td>2D</td>
<td>29</td>
<td>C5</td>
<td>89</td>
<td>6F</td>
<td>B7</td>
<td>62</td>
<td>2E</td>
<td>AA</td>
<td>18</td>
<td>BE</td>
</tr>
<tr>
<td>B</td>
<td>FC</td>
<td>56</td>
<td>3E</td>
<td>4B</td>
<td>C6</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>
<td>5A</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>B1</td>
<td>12</td>
<td>10</td>
<td>59</td>
<td>27</td>
<td>80</td>
<td>EC</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>E5</td>
<td>7A</td>
<td>9F</td>
<td>93</td>
<td>C9</td>
<td>9C</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>C5</td>
<td>EB</td>
<td>BB</td>
<td>3C</td>
<td>83</td>
<td>53</td>
<td>99</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>26</td>
<td>E1</td>
<td>69</td>
<td>14</td>
<td>63</td>
<td>55</td>
<td>21</td>
<td>0C</td>
</tr>
</tbody>
</table>
For the third row, a 3-byte circular left shift is performed. The following is an example of ShiftRows:

\[
\begin{array}{cccc}
87 & F2 & 4D & 97 \\
EC & 6E & 4C & 90 \\
4A & C3 & 46 & E7 \\
8C & D8 & 95 & A6 \\
\end{array} \quad \longrightarrow \quad \begin{array}{cccc}
87 & F2 & 4D & 97 \\
6E & 4C & 90 & EC \\
46 & E7 & 4A & C3 \\
A6 & 8C & D8 & 95 \\
\end{array}
\]

The inverse shift row transformation, called InvShiftRows, performs the circular shifts in the opposite direction for each of the last three rows, with a one-byte circular right shift for the second row, and so on.

Figure 3. AES Row and Column Operations.
The shift row transformation is more substantial than it may first appear. This is because the State, as well as the cipher input and output, is treated as an array of four 4-byte columns. Thus, on encryption, the first four bytes of the plaintext are copied to the first column of State, and so on. Further, as will be seen, the round key is applied to State column by column. Thus, a row shift moves an individual byte from one column to another, which is a linear distance of a multiple of 4 bytes. Also note that the transformation ensures that the four bytes of one column are spread out to four different columns.

Mix Column Transformation

The forward mix column transformation, called MixColumns, operates on each column individually. Each byte of a column is mapped into a new value that is a function of all four bytes in the column. The transformation can be defined by the following matrix multiplication on State (Figure 3b):

\[
\begin{bmatrix}
02 & 03 & 01 & 01 \\
01 & 02 & 03 & 01 \\
01 & 01 & 02 & 03 \\
03 & 01 & 01 & 02
\end{bmatrix}
\begin{bmatrix}
s_0,0 & s_0,1 & s_0,2 & s_0,3 \\
s_1,0 & s_1,1 & s_1,2 & s_1,3 \\
s_2,0 & s_2,1 & s_2,2 & s_2,3 \\
s_3,0 & s_3,1 & s_3,2 & s_3,3
\end{bmatrix}
= \begin{bmatrix}
s'_0,0 & s'_0,1 & s'_0,2 & s'_0,3 \\
s'_1,0 & s'_1,1 & s'_1,2 & s'_1,3 \\
s'_2,0 & s'_2,1 & s'_2,2 & s'_2,3 \\
s'_3,0 & s'_3,1 & s'_3,2 & s'_3,3
\end{bmatrix}
\]

Each element in the product matrix is the sum of products of elements of one row and one column. In this case, the individual additions and multiplications\(^3\) are performed in \(GF(2^8)\). The MixColumns transformation on a single column \(c\) (\(0 \leq c \leq 3\)) of State can be expressed as:

\[
\begin{align*}
s'_0,c &= (2 \cdot s_{0,c}) \oplus (3 \cdot s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\
\end{align*}
\]

\[
\begin{align*}
s'_1,c &= s_{0,c} \oplus (2 \cdot s_{1,c}) \oplus (3 \cdot s_{2,c}) \oplus s_{3,c} \\
s'_2,c &= s_{0,c} \cdot s_{1,c} \oplus (2 \cdot s_{2,c}) \oplus (3 \cdot s_{3,c}) \\
s'_3,c &= (3 \cdot s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (2 \cdot s_{3,c})
\end{align*}
\]

The following is an example of MixColumns:

<table>
<thead>
<tr>
<th>87</th>
<th>F2</th>
<th>4D</th>
<th>97</th>
</tr>
</thead>
<tbody>
<tr>
<td>6E</td>
<td>4C</td>
<td>90</td>
<td>EC</td>
</tr>
<tr>
<td>46</td>
<td>E7</td>
<td>4A</td>
<td>C3</td>
</tr>
<tr>
<td>A6</td>
<td>8C</td>
<td>D8</td>
<td>95</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccc}
47 & 40 & A3 & 4C \\
37 & D4 & 70 & 9F \\
94 & E4 & 3A & 42 \\
ED & A5 & A6 & BC
\end{array}
\]

Let us verify the first column of this example. In \(GF(2^8)\), addition is the bitwise XOR operation. Multiplication can be performed according to the following rule: multiplication of a value by \{02\} can be implemented as a one-bit

\(^3\)We follow the convention of FIPS PUB 197 and use the symbol \(\cdot\) to indicate multiplication over the finite field \(GF(2^8)\) and \(\oplus\) to indicate bitwise XOR, which corresponds to addition in \(GF(2^8)\).
left shift followed by a conditional bitwise XOR with (00011011) if the leftmost bit of the original value (prior to the shift) is 1. Thus, to verify the MixColumns transformation on the first column, we need to show that:

\[
\begin{align*}
(\{02\} \cdot \{87\}) & \oplus (\{03\} \cdot \{6E\}) \oplus \{46\} \oplus \{A6\} = \{47\} \\
\{87\} & \oplus (\{02\} \cdot \{6E\}) \oplus (\{03\} \cdot \{46\}) \oplus \{A6\} = \{37\} \\
\{02\} \cdot \{87\} & \oplus \{6E\} \oplus (\{02\} \cdot \{46\}) \oplus (\{03\} \cdot \{A6\}) = \{94\} \\
(\{03\} \cdot \{87\}) & \oplus \{6E\} \oplus \{46\} \oplus (\{02\} \cdot \{A6\}) = \{ED\}
\end{align*}
\]

For the first equation, we have \(\{02\} \cdot \{87\} = (0000\ 1110) \oplus (0001\ 1011) = (0001\ 0101)\); and \(\{03\} \cdot \{6E\} = \{6E\} \oplus (\{02\} \cdot \{6E\}) = (0110\ 1110)(1101\ 1100) = (1011\ 0010)\). Then,

\[
\begin{align*}
\{02\} \cdot \{87\} &= 00010101 \\
\{03\} \cdot \{6E\} &= 10110010 \\
\{46\} &= 01000110 \\
\{A6\} &= 10100110 \\
01000111 &= \{47\}
\end{align*}
\]

The other equations can be similarly verified.

The inverse mix column transformation, called InvMixColumns, is defined by the following matrix multiplication:

\[
\begin{bmatrix}
0E & 0B & 0D & 09 \\
09 & 0E & 0B & 0D \\
0D & 09 & 0E & 0B \\
0B & 0D & 09 & 0E
\end{bmatrix}
\begin{bmatrix}
s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\
s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\
s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\
s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3}
\end{bmatrix}
= \begin{bmatrix}
s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\
s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\
s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\
s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3}
\end{bmatrix}
\tag{5}
\]

It is not immediately clear that Equation (5) is the inverse of Equation (3). What we need to show is that:

\[
\begin{bmatrix}
0E & 0B & 0D & 09 \\
09 & 0E & 0B & 0D \\
0D & 09 & 0E & 0B \\
0B & 0D & 09 & 0E
\end{bmatrix}
\begin{bmatrix}
02 & 03 & 01 & 01 \\
01 & 02 & 03 & 01 \\
01 & 01 & 02 & 03 \\
03 & 01 & 01 & 02
\end{bmatrix}
\begin{bmatrix}
s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\
s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\
s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\
s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3}
\end{bmatrix}
= \begin{bmatrix}
s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\
s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\
s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\
s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3}
\end{bmatrix}
\]

176
which is equivalent to showing that:

\[
\begin{bmatrix}
0E & 0B & 0D & 09 \\
09 & 0E & 0B & 0D \\
0D & 09 & 0E & 0B \\
0B & 0D & 09 & 0E \\
\end{bmatrix}
\begin{bmatrix}
02 & 03 & 01 & 01 \\
01 & 02 & 03 & 01 \\
01 & 01 & 02 & 03 \\
03 & 01 & 01 & 02 \\
\end{bmatrix}
= \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\] (6)

That is, the inverse transformation matrix times the forward transformation matrix equals the identity matrix. To verify the first column of Equation (6), we need to show that:

\[
\begin{align*}
(\{0E\} \oplus \{02\}) \oplus \{0B\} \oplus \{0D\} \oplus (\{09\} \cdot \{03\}) &= \{01\} \\
(\{09\} \oplus \{02\}) \oplus \{0E\} \oplus \{0B\} \oplus (\{0D\} \cdot \{03\}) &= \{00\} \\
(\{0D\} \oplus \{02\}) \oplus \{09\} \oplus \{0E\} \oplus (\{0B\} \cdot \{03\}) &= \{00\} \\
(\{0B\} \oplus \{02\}) \oplus \{0D\} \oplus \{09\} \oplus (\{0E\} \cdot \{03\}) &= \{00\}
\end{align*}
\]

For the first equation, we have \(\{0E\} \cdot \{02\} = 00011100\); and \(\{09\} \cdot \{03\} = \{09\} \oplus (\{09\} \cdot \{02\}) = 00001001 \cdot 00010010 = 00011011\). Then,

\[
\begin{align*}
\{0E\} \cdot \{02\} &= 00011100 \\
\{0B\} &= 00001011 \\
\{0D\} &= 00001101 \\
\{09\} \cdot \{03\} &= 00011111 \\
& \quad \text{00000001}
\end{align*}
\]

The other equations can be similarly verified.

The coefficients of the matrix in Equation (3) are based on a linear code with maximal distance between code words, which ensures a good mixing among the bytes of each column. The mix column transformation combined with the shift row transformation ensures that after a few rounds, all output bits depend on all input bits. See [2] for a discussion.

In addition, the choice of coefficients in MixColumns, which are all 01, 02, or 03, was influenced by implementation considerations. As was discussed, multiplication by these coefficients involves at most a shift and an XOR. The coefficients in InvMixColumns are more formidable to implement. However, encryption was deemed more important than encryption for two reasons:

1. Two common modes of operation for symmetric block ciphers, cipher feedback mode (CFB) and output feedback mode (OFB), use only the encryption algorithm for both encryption and decryption.
2. As with any block cipher, AES can be used to construct a message authentication code, and for this only encryption is used.

Add Round Key Transformation

In the forward add round key transformation, called AddRoundKey, the 128 bits of State are bitwise XORed with the 128 bits of the round key. The operation is viewed as a column-wise operation between the four bytes of a State column and one word of the round key; it can also be viewed as a byte-level operation. The following is an example of AddRoundKey:

\[
\begin{array}{cccc}
47 & 40 & A3 & 4C \\
37 & D4 & 70 & 9F \\
94 & E4 & 3A & 42 \\
ED & A5 & A6 & BC \\
\end{array} \oplus \begin{array}{cccc}
AC & 19 & 28 & 57 \\
77 & FA & D1 & 5C \\
66 & DC & 29 & 00 \\
F3 & 21 & 41 & 6A \\
\end{array} = \begin{array}{cccc}
EB & 59 & 8B & 1B \\
40 & 2E & A1 & C3 \\
F2 & 38 & 13 & 42 \\
1E & 84 & E7 & D2 \\
\end{array}
\]

The first matrix is State, and the second matrix is the round key.

The inverse add round key transformation is identical to the forward add round key transformation, because the XOR operation is its own inverse.

The add round key transformation is as simple as possible and affects every bit of State. The complexity of the round key expansion, plus the complexity of the other stages of AES, ensure security.

AES Key Expansion

The AES key expansion algorithm takes as input a 4-word (16-byte) key and produces a linear array of 44 words (156 bytes). This is sufficient to provide a 4-word round key for the initial Add Round Key stage and each of the 10 rounds of the cipher. The following pseudocode describes the expansion:

```
KeyExpansion (byte key[16], word w[44])
{
    word temp
    for (i = 0; i < 4; i++) w[i] = (key[4*i], key[4*i+1], key[4*i+2], key[4*i+3]);
    for (i = 4; i < 44; i++)
    {
        temp = w[i - 1];
        if (i mod 4 = 0) temp = SubWord (RotWord (temp)) ⊕ Rcon[i/4];
        w[i] = w[i-4] ⊕ temp
    }
}
```
The key is copied into the first four words of the expanded key. The remainder of the expanded key is filled in four words at a time. Each added word \( w[i] \) depends on the immediately preceding word, \( w[i-1] \), and the word four positions back, \( w[i-4] \). In three out of four cases, a simple XOR is used. For a word whose position in the \( w \) array is a multiple of 4, a more complex function is used. Figure 4 illustrates the generation of the first eight words of the expanded key, using the symbol \( g \) to represent that complex function. The function \( g \) consists of the following subfunctions:

1. RotWord performs a one-byte circular left shift on a word. This means that an input word \( [b_0, b_1, b_2, b_3] \) is transformed into \( [b_1, b_2, b_3, b_0] \).
2. SubWord performs a byte substitution on each byte of its input word, using the S-box (Table 2a).
3. The result of steps 1 and 2 is XORed with a round constant, Rcon[j].

The round constant is a word in which the three rightmost bytes are always 0. Thus the effect of an XOR of a word with Rcon is to only perform an XOR on the leftmost byte of the word. The round constant is different for each round and is defined as \( \text{Rcon}[j] = (\text{RC}[j], 0, 0, 0) \), with \( \text{RC}[1] = 1 \), \( \text{RC}[j] = 2 \cdot \text{RC}[j-1] \) and with multiplication defined over the field \( GF(2^8) \). The values of \( \text{RC}[j] \) in hexadecimal are:

179
For example, suppose that the round key for round 8 is:

EA D2 73 21 B5 8D BA D2 31 2B F5 60 7F 8D 29 2F

Then the first four bytes (first column) of the round key for round 9 are calculated as follows:

<table>
<thead>
<tr>
<th>i</th>
<th>temp</th>
<th>After RotWord</th>
<th>After SubWord</th>
<th>Rcon (9)</th>
<th>After XOR with Rcon</th>
<th>w[i - 4]</th>
<th>w[i] = temp ⊕ w[i - 4]</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>7F8D292F</td>
<td>8D292F7F</td>
<td>5DA515D2</td>
<td>1B000000</td>
<td>46A515D2</td>
<td>EAD27321</td>
<td>AC7766F3</td>
</tr>
</tbody>
</table>

The Rijndael developers designed the expansion key algorithm to be resistant to known cryptanalytic attacks. The inclusion of a round-dependent round constant eliminates the symmetry, or similarity, between the way in which round keys are generated in different rounds. The specific criteria used are [2]:

- An invertible transformation, i.e., knowledge of any \( N_k \) consecutive words of the Expanded Key enables regeneration the entire expanded key (\( N_k = \) key size in words)
- Speed on a wide range of processors;
- Usage of round constants to eliminate symmetries;
- Diffusion of cipher key differences into the round keys; that is, each key bit affects many round key bits
- Knowledge of a part of the cipher key or round key does not enable calculation of many other round key bits
- Enough non-linearity to prohibit the full determination of round key differences from cipher key differences only
- Simplicity of description

Equivalent Inverse Cipher

As was mentioned, the AES decryption cipher is not identical to the encryption cipher (Figure 1). That is, the sequence of transformations for decryption differs from that for encryption, although the form of the key schedules for encryption and decryption is the same. This has the disadvantage that two separate software or firmware modules are needed for applications that require both encryption and decryption. There is, however, an equivalent version of the decryption algorithm that has the same structure as the encryption algorithm. The equivalent version
has the same sequence of transformations as the encryption algorithm (with transformations replaced by their inverses). To achieve this equivalence, a change in key schedule is needed.

Two separate changes are needed to bring the decryption structure in line with the encryption structure. An encryption round has the structure SubBytes, ShiftRows, MixColumns, AddRoundKey. The standard decryption round has the structure InvShiftRows, InvSubBytes, AddRoundKey, InvMixColumns. Thus, the first two stages of the decryption round need to be interchanged, and the second two changes of the decryption round need to be interchanged.

InvShiftRows affects the sequence of bytes in State, but does not alter byte contents and does not depend on byte contents to perform its transformation. InvSubBytes affects the contents of bytes in State, but does not alter byte sequence and does not depend on byte sequence to perform its transformation. Thus, these two operations commute and can be interchanged. For a given State $S_i$,

$$\text{InvShiftRows}[\text{InvSubBytes}(S_i)] = \text{InvSubBytes}[\text{InvShiftRows}(S_i)].$$

The transformations AddRoundKey and InvMixColumns do not alter the sequence of bytes in State. If we view the key as a sequence of words, then both AddRoundKey and InvMixColumns operate on State one column at a time. These two operations are linear with respect to the column input. That is, for a given State $S_i$ and a given round key $w_j$,

$$\text{InvMixColumns}(S_i \oplus w_j) = [\text{InvMixColumns}(S_i)] \oplus [\text{InvMixColumns}(w_j)].$$

To see this, suppose that the first column of State $S_i$ is the sequence $(y_0, y_1, y_2, y_3)$ and the first column of the round key $w_j$ is $(k_0, k_1, k_2, k_3)$. Then we need to show that

$$\begin{bmatrix}
0E & 0B & 0D & 09 \\
09 & 0E & 0B & 0D \\
0D & 09 & 0E & 0B \\
0B & 0D & 09 & 0E
\end{bmatrix}
\begin{bmatrix}
y_0 \oplus k_0 \\
y_1 \oplus k_1 \\
y_2 \oplus k_2 \\
y_3 \oplus k_3
\end{bmatrix}
= 
\begin{bmatrix}
0E & 0B & 0D & 09 \\
09 & 0E & 0B & 0D \\
0D & 09 & 0E & 0B \\
0B & 0D & 09 & 0E
\end{bmatrix}
\begin{bmatrix}
y_0 \\
y_1 \\
y_2 \\
y_3
\end{bmatrix}$$

$$\oplus
\begin{bmatrix}
0E & 0B & 0D & 09 \\
09 & 0E & 0B & 0D \\
0D & 09 & 0E & 0B \\
0B & 0D & 09 & 0E
\end{bmatrix}
\begin{bmatrix}
k_0 \\
k_1 \\
k_2 \\
k_3
\end{bmatrix}$$

181
Let us demonstrate that for the first column entry. We need to show that:

\[
(0E \cdot (y_0 \oplus k_0)) \oplus (0B \cdot (y_1 \oplus k_1)) \oplus (0D \cdot (y_2 \oplus k_2)) \oplus (09 \cdot (y_3 \oplus k_3)) \\
= (0E \cdot y_0) \oplus (0B \cdot y_1) \oplus (0D \cdot y_2) \oplus (09 \cdot y_3) \\
\oplus (0E \cdot k_0) \oplus (0B \cdot k_1) \oplus (0D \cdot k_2) \oplus (09 \cdot k_3)
\]

This equation is valid by inspection. Thus, we can interchange AddRoundKey and InvMixColumns, provided that we first apply InvMixColumns to the round key. Note that we do not need to apply InvMixColumns to the round key for the input to the first AddRoundKey transformation (preceding the first round) nor to the last AddRoundKey transformation (in round 10). This is because these two AddRoundKey transformations are not interchanged with InvMixColumns to produce the equivalent decryption algorithm.

To summarize, an encryption round has the structure SubBytes, ShiftRows, MixColumns, AddRoundKey. the standard decryption round has the structure InvShiftRows, InvSubBytes, AddRoundKey, InvMixColumns. The equivalent decryption round has the structure InvSubBytes, InvShiftRows, InvMixColumns, AddRoundKey.

**Implementation Aspects**

The Rijndael proposal [2] provides some suggestions for efficient implementation on 8-bit processors, typical for current smart cards, and on 32-bit processors, typical for PCs.

AES can be implemented very efficiently on an 8-bit processor. AddRoundKey is a byte-wise XOR operation. ShiftRows is a simple byte shifting operation. SubBytes operates at the byte level and only requires a table of 256 bytes.

The transformation MixColumns requires matrix multiplication in the field \(GF(2^8)\), which means that all operations are carried out on bytes. MixColumns only requires multiplication by \{02\} and \{03\}, which, as we have seen, involved simple shifts, conditional XORs and XORs. This can be implemented in a more efficient way that eliminates the shifts and conditional XORs. Equation Set (4) shows the equations for the MixColumns transformation on a single column. Using the identity \(\{03\} \oplus x = (\{02\} \oplus x) \oplus x\), we rewrite Equation Set (4):

\[
\begin{align*}
T_{mp} &= s_{0,j} \oplus s_{1,j} \oplus s_{2,j} \oplus s_{3,j} \\
\left(s_{0,j}ight)' &= s_{0,j} \oplus T_{mp} \oplus [2 \cdot (s_{0,j} \oplus s_{1,j})] \\
\left(s_{1,j}ight)' &= s_{1,j} \oplus T_{mp} \oplus [2 \cdot (s_{1,j} \oplus s_{2,j})] \\
\left(s_{2,j}ight)' &= s_{2,j} \oplus T_{mp} \oplus [2 \cdot (s_{2,j} \oplus s_{3,j})] \\
\left(s_{3,j}ight)' &= s_{3,j} \oplus T_{mp} \oplus [2 \cdot (s_{3,j} \oplus s_{4,j})]
\end{align*}
\]
Equation Set (7) is verified by expanding and eliminating terms.

The multiplication by \{02\} involves a shift and a conditional XOR. Such an implementation may be vulnerable to a timing attack. To counter this attack, and also to increase processing efficiency at the cost of some storage, the multiplication can be replaced by a table lookup. Define the 256-byte table \(X2\), such that \(X2[i] = \{02\} \cdot i\). Then Equation Set (7) can be rewritten as:

\[
\begin{align*}
\text{Tmp} & = s_{0,j} \oplus s_{1,j} \oplus s_{2,j} \oplus s_{3,j} \\
\text{s}'_{0,j} & = s_{0,j} \oplus \text{Tmp} \oplus X2[s_{0,j} \oplus s_{1,j}] \\
\text{s}'_{1,j} & = s_{1,j} \oplus \text{Tmp} \oplus X2[s_{1,j} \oplus s_{2,j}] \\
\text{s}'_{2,j} & = s_{2,j} \oplus \text{Tmp} \oplus X2[s_{2,j} \oplus s_{3,j}] \\
\text{s}'_{3,j} & = s_{3,j} \oplus \text{Tmp} \oplus X2[s_{3,j} \oplus s_{0,j}]
\end{align*}
\]

The implementation just described uses only 8-bit operations. For a 32-bit processor, a more efficient implementation can be achieved if operations are defined on 32-bit words. To show this, we first define the four transformations of a round in algebraic form. Suppose we begin with a \textit{State} matrix with elements \(a_{i,j}\) and a round key matrix with elements \(k_{i,j}\) Then, the transformations can be expressed as follows:

<table>
<thead>
<tr>
<th>SubBytes</th>
<th>( b_{i,j} = S[a_{i,j}] )</th>
</tr>
</thead>
</table>
| ShiftRows | \[
\begin{bmatrix}
  c_{0,j} \\
  c_{1,j} \\
  c_{2,j} \\
  c_{3,j}
\end{bmatrix} = \begin{bmatrix}
  b_{0,j} \\
  b_{1,j-1} \\
  b_{2,j-2} \\
  b_{3,j-3}
\end{bmatrix}
\]
| MixColumns | \[
\begin{bmatrix}
  d_{0,j} \\
  d_{1,j} \\
  d_{2,j} \\
  d_{3,j}
\end{bmatrix} = \begin{bmatrix}
  02 & 03 & 01 & 01 \\
  01 & 02 & 03 & 01 \\
  01 & 01 & 02 & 03 \\
  03 & 01 & 01 & 02
\end{bmatrix} \begin{bmatrix}
  c_{0,j} \\
  c_{1,j} \\
  c_{2,j} \\
  c_{3,j}
\end{bmatrix}
\]
| AddRoundKey | \[
\begin{bmatrix}
  e_{0,j} \\
  e_{1,j} \\
  e_{2,j} \\
  e_{3,j}
\end{bmatrix} = \begin{bmatrix}
  d_{0,j} \\
  d_{1,j} \\
  d_{2,j} \\
  d_{3,j}
\end{bmatrix} \oplus \begin{bmatrix}
  k_{0,j} \\
  k_{1,j} \\
  k_{2,j} \\
  k_{3,j}
\end{bmatrix}
\]

In the ShiftRows equation, the column indices are taken mod 4. We can combine all of these expressions into a single equation:
\[
\begin{bmatrix}
  e_{0,j} \\
  e_{1,j} \\
  e_{2,j} \\
  e_{3,j}
\end{bmatrix}
= 
\begin{bmatrix}
  02 & 03 & 01 & 01 \\
  01 & 02 & 03 & 01 \\
  01 & 01 & 02 & 03 \\
  03 & 01 & 01 & 02
\end{bmatrix}
\begin{bmatrix}
  S[a_{0,j}] \\
  S[a_{1,j-1}] \\
  S[a_{2,j-2}] \\
  S[a_{3,j-3}]
\end{bmatrix}
\oplus
\begin{bmatrix}
  k_{0,j} \\
  k_{1,j} \\
  k_{2,j} \\
  k_{3,j}
\end{bmatrix}
\]

In the second equation, we are expressing the matrix multiplication as a linear combination of vectors. Now, we define four 256-word (1024-byte) tables, as follows:

\[
T_0[x] = 
\begin{bmatrix}
  02 & \cdot S[x] \\
  01 & \cdot S[x] \\
  01 & \cdot S[x] \\
  03 & \cdot S[x]
\end{bmatrix}
\]

\[
T_1[x] = 
\begin{bmatrix}
  03 & \cdot S[x] \\
  02 & \cdot S[x] \\
  01 & \cdot S[x] \\
  01 & \cdot S[x]
\end{bmatrix}
\]

\[
T_2[x] = 
\begin{bmatrix}
  01 & \cdot S[x] \\
  03 & \cdot S[x] \\
  02 & \cdot S[x] \\
  01 & \cdot S[x]
\end{bmatrix}
\]

\[
T_3[x] = 
\begin{bmatrix}
  01 & \cdot S[x] \\
  01 & \cdot S[x] \\
  03 & \cdot S[x] \\
  02 & \cdot S[x]
\end{bmatrix}
\]

Thus, each table takes as input a byte value and produces a column vector (a 32-bit word) that is a function of the S-box entry for that byte value. These tables can be calculated in advance.

We can define a round function operating on a column in the following fashion:

\[
\begin{bmatrix}
  s'_{0,j} \\
  s'_{1,j} \\
  s'_{2,j} \\
  s'_{3,j}
\end{bmatrix}
= T_0[s_{0,j}] \oplus T_1[s_{1,j-1}] \oplus T_2[s_{2,j-2}] \oplus T_3[s_{3,j-3}] \oplus
\begin{bmatrix}
  k_{0,j} \\
  k_{1,j} \\
  k_{2,j} \\
  k_{3,j}
\end{bmatrix}
\]

As a result, and implementation based on the preceding equation requires only 4 table lookups and 4 XORs per column per round, plus 4 Kbytes to store...
the table. The developers of Rijndael believe that this compact, efficient implementation was probably one of the most important factors in the selection of Rijndael for AES.

APPENDIX: JUST ENOUGH FINITE FIELD THEORY

We present here the basic concepts of finite fields. For more detail see [7]; for a thorough treatment, see [5]. A field is a set of elements \( F \), together with two binary operations \( (+, \cdot) \), called addition and multiplication, such that for all \( a, b, c \) in \( F \) the following axioms are obeyed (multiplication is represented here by concatenation):

(A1) Closure: If \( a \) and \( b \) belong to \( G \), then \( a + b \) is also in \( G \).

(A2) Associative: \( a + (b + c) = (a + b) + c \) for all \( a, b, c \) in \( G \).

(A3) Identity element: There is an element \( e \) in \( G \) such that \( a + e = e + a = a \) for all \( a \) in \( G \).

(A4) Inverse element: For each \( a \) in \( G \) there is an element \( a' \) in \( G \) such that \( a + a' = a' + a = e \).

(A5) Commutative: \( a + b = b + a \) for all \( a, b \) in \( G \).

(M1) Closure under multiplication: If \( a \) and \( b \) belong to \( R \), then \( ab \) is also in \( R \).

(M2) Associativity of multiplication: \( a(bc) = (ab)c \) for all \( a, b, c \) in \( R \).

(M3) Distributive laws: \( a(b + c) = ab + ac \) for all \( a, b, c \) in \( R \).

(M4) Commutativity of multiplication: \( ab = ba \) for all \( a, b \) in \( R \).

(M5) Multiplicative identity: There is an element \( 1 \) in \( R \) such that \( a1 = 1a = a \) for all \( a \) in \( R \).

(M6) No zero divisors: If \( a, b \) in \( R \) and \( ab = 0 \), then either \( a = 0 \) or \( b = 0 \).

(M7) Multiplicative inverse: For each \( a \) in \( F \), except 0, there is an element \( a^{-1} \) in \( F \) such that \( aa^{-1} = (a^{-1})a = 1 \).

In essence, a field is a set in which we can do addition, subtraction, multiplication, and division without leaving the set. Division is defined with the following rule: \( a/b = a(b^{-1}) \). Familiar examples of fields are the rational numbers, the real numbers, and the complex numbers. The set of all integers is not a field,
because not every element of the set has a multiplicative inverse; in fact, only the elements 1 and -1 have multiplicative inverses in the integers.

An example of a finite field (one with a finite number of elements) is the set $\mathbb{Z}_p$ consisting of all the integers $\{0, 1, \ldots, p - 1\}$, where $p$ is a prime number, and in which arithmetic is carried out modulo $p$.

Virtually all encryption algorithms, both conventional and public-key, involve arithmetic operations on integers. If one of the operations that is used in the algorithm is division, then we need to work in arithmetic defined over a field; this is because division requires that each nonzero element have a multiplicative inverse. For convenience and for implementation efficiency, we would also like to work with integers that fit exactly into a given number of bits, with no wasted bit patterns. That is, we wish to work with integers in the range 0 through $2^n - 1$, which fit into an $n$-bit word. Unfortunately the set of such integers, $\mathbb{Z}_{2^n}$, using modular arithmetic, is not a field. For example, the integer 2 has no multiplicative inverse, that is, there is no integer $b$, such that $2b \mod 2^n = 1$.

There is a way of defining a finite field containing $2^n$ elements; such a field is referred to as $GF(2^n)$. $GF$ stands for Galois field, in honor of the mathematician who first studied finite fields. Consider the set, $S$, of all polynomials of degree $n - 1$ or less with binary coefficients. Thus, each polynomial has the form:

$$f(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1x + a_0 = \sum_{i=0}^{n-1} a_i x^i$$

where each $a_i$ takes on the value 0 or 1. There are a total of $2^n$ different polynomials in $S$. For $n = 3$, the $2^3 = 8$ polynomials in the set are:

$$\begin{array}{cccc}
0 & x & x^2 & x^2 + x \\
1 & x + 1 & x^2 + 1 & x^2 + x + 1 \\
\end{array}$$

With the appropriate definition of arithmetic operations, each such set $S$ is a finite field. The definition consists of the following elements:

1. Arithmetic follows the ordinary rules of polynomial arithmetic using the basic rules of algebra, with the following two refinements.

2. Arithmetic on the coefficients is performed modulo 2. This is the same as the XOR operation.

3. If multiplication results in a polynomial of degree greater than $n - 1$, then the polynomial is reduced modulo some irreducible polynomial $m(x)$ of degree $n$. That is, we divide by $m(x)$ and keep the remainder. For a polynomial $f(x)$, the remainder is expressed as $r(x) = f(x) \mod m(x)$. A polynomial $m(x) \mod m(x)$
is called irreducible if and only if \( m(x) \) cannot be expressed as a product of
two polynomials, both of degree lower than that of \( m(x) \).

For example, to construct the finite field \( GF(2^3) \), we need to choose an irre-
ducible polynomial of degree 3. There are only two such polynomials: \( x^3+x^2+1 \)
and \( x^3 + x + 1 \). Using the latter, Table 3 shows the multiplication table
for \( GF(2^3) \). Addition is equivalent to taking the XOR of like terms. Thus
\((x+1) + x = 1\).

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x+1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x^2</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x^3</td>
<td>0</td>
<td>x^2</td>
<td>x^2</td>
<td>x^2</td>
<td>x^2</td>
<td>x^2</td>
<td>x^2</td>
<td>x^2</td>
</tr>
<tr>
<td>x^4</td>
<td>0</td>
<td>x^3</td>
<td>x^3</td>
<td>x^3</td>
<td>x^3</td>
<td>x^3</td>
<td>x^3</td>
<td>x^3</td>
</tr>
<tr>
<td>x^5</td>
<td>0</td>
<td>x^4</td>
<td>x^4</td>
<td>x^4</td>
<td>x^4</td>
<td>x^4</td>
<td>x^4</td>
<td>x^4</td>
</tr>
<tr>
<td>x^6</td>
<td>0</td>
<td>x^5</td>
<td>x^5</td>
<td>x^5</td>
<td>x^5</td>
<td>x^5</td>
<td>x^5</td>
<td>x^5</td>
</tr>
<tr>
<td>x^7</td>
<td>0</td>
<td>x^6</td>
<td>x^6</td>
<td>x^6</td>
<td>x^6</td>
<td>x^6</td>
<td>x^6</td>
<td>x^6</td>
</tr>
<tr>
<td>x^8</td>
<td>0</td>
<td>x^7</td>
<td>x^7</td>
<td>x^7</td>
<td>x^7</td>
<td>x^7</td>
<td>x^7</td>
<td>x^7</td>
</tr>
<tr>
<td>x^9</td>
<td>0</td>
<td>x^8</td>
<td>x^8</td>
<td>x^8</td>
<td>x^8</td>
<td>x^8</td>
<td>x^8</td>
<td>x^8</td>
</tr>
<tr>
<td>x^{10}</td>
<td>0</td>
<td>x^9</td>
<td>x^9</td>
<td>x^9</td>
<td>x^9</td>
<td>x^9</td>
<td>x^9</td>
<td>x^9</td>
</tr>
<tr>
<td>x^{11}</td>
<td>0</td>
<td>x^{10}</td>
<td>x^{10}</td>
<td>x^{10}</td>
<td>x^{10}</td>
<td>x^{10}</td>
<td>x^{10}</td>
<td>x^{10}</td>
</tr>
</tbody>
</table>

Table 3. Polynomial Multiplication Modulo \((x^3 + x + 1)\).

A polynomial in \( GF(2^n) \) can be uniquely represented by its \( n \) binary coef-
ficients \((a_{n-1}a_{n-2}...a_0)\). Therefore, every polynomial in \( GF(2^n) \) can be re-
presented by an \( n \)-bit number. Addition is performed by taking the bitwise XOR of
the two \( n \)-bit elements. There is no simple XOR operation that will accomplish
multiplication in \( GF(2^n) \). However, a reasonably straightforward, easily imple-
mented, technique is available. In essence, it can be shown that multiplication of
a number in \( GF(2^n) \) by 2 consists of a left shift followed by a conditional XOR
with a constant. Multiplication by larger numbers can be achieved by repeated
application of this rule.

For example, AES uses arithmetic in the finite field \( GF(2^8) \), with the irre-
ducible polynomial \( m(x) = x^8 + x^4 + x^3 + x + 1 \). Consider two elements
\( A = (a_7a_6...a_1a_0) \) and \( B = (b_7b_6...b_1b_0) \). The sum \( A + B = (c_7c_6...c_1c_0) \),
where \( c_i = a_i \oplus b_i \). The multiplication \( \{02\} \cdot A \) equals \((a_6...a_1a_00)\) if \( a_7 = 0 \),
and equals \((a_6...a_1a_00) \oplus (00011011)\) if \( a_7 = 1 \).

REFERENCES


187


**BIOGRAPHICAL SKETCH**

William Stallings holds a PhD from MIT in Computer Science and a BS from Notre Dame in electrical engineering. He has authored numerous books on security, computer networking, and computer architecture. He has five times received the award for the best Computer Science and Engineering textbook of the year from the Textbook and Academic Authors Association. His most recent book is *Cryptography and Network Security, Third Edition* (Prentice Hall, 2002). He created and maintains the Computer Science Student Resource Site at [WilliamStallings.com/StudentSupport.html](http://WilliamStallings.com/StudentSupport.html). This site provides documents and links on a variety of subjects of general interest to computer science students (and professionals).