how it works

THE ADVANCED ENCRYPTION STANDARD IS GAINING STEAM AS A STRONGER ALTERNATIVE TO THE DATA ENCRYPTION STANDARD. NEXT-GENERATION APPLICATIONS WILL GO BEYOND SECURE NETWORKING PROTOCOLS TO INCLUDE SMART CARDS AND ELECTRONIC-MEDIA-CONTENT PROTECTION.

Encryption and security: the Advanced Encryption Standard

By Stuart Allman, Cypress Semiconductor

HISTORY HAS SHOWN that it is always necessary for security to evolve to meet the demands of new technology. Moore's Law may work wonders for computing horsepower, but it creates nightmares for security specialists. In recent years, the DES (Data Encryption Standard) has become vulnerable to attack by the raw computing horsepower that even a garage hobbyist possesses. Most security applications now use a method called 3DES (triple-DES) to extend the cipher strength of DES.

The NIST (National Institute of Standards and Technology) realizes that DES may no longer be appropriate for some high-security applications. In response to this need for better cipher strength, NIST commissioned a 1997 study of the AES (Advanced Encryption Standard). The final selection of this study was an application of the Rijndael (pronounced rhine-doll) algorithm. Like DES, the AES acts as a block cipher, but with much larger key lengths and improved cipher strength.

WHAT IS THE AES?

The AES algorithm converts a block of 128 bits (referred to as plain text) to a 128-bit block of encrypted data (referred to as cipher text). In the sense that they are both block ciphers, the AES algorithm is just like the DES algorithm (in standard ECB, or Electronic Code Book, mode).

The algorithm uses one of three cipher key strengths: a 128-, 192-, or 256-bit encryption key. Each encryption-key size causes the algorithm to behave slightly differently. So, the increasing key sizes not only offer a larger number of bits with which you can scramble the data, but also increase the complexity of the cipher algorithm.

The AES algorithm repeats its core a number of times, depending on the encryption-key size. Just like DES, the AES algorithm refers to these loop repetitions as "rounds," and the cipher must also complete preround and postround operations. However, unlike DES, the AES algorithm is not truly symmetric. The operations for the reverse cipher are the inverse of the forward-cipher operations, and they are performed in a different order.

The following terms are commonly used to describe the AES algorithm. It is a good idea to become familiar with these terms before reading additional literature or the FIPS (Federal Information Processing Standards) specification (**Reference 1**).

- A *round* is an iteration of the main part of the AES algorithms. AES algorithms contain a variable number of rounds, depending on the key size.

PLAIN/CIPHER TEXT (ARRAY OF 16 BYTES)

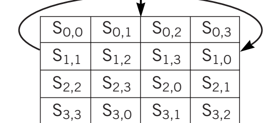
$b_0, b_1, b_2, b_3, b_4, b_5, \ldots b_{15}$

STATE MATRIX (4×4 MATRIX OF BYTES)

ROW_0	$S_{0,0}=b_0$	b_4	b_8	b_{12}
ROW_1	b_1	b_5	b_9	b_{13}
ROW_2	b_2	b_6	b_{10}	b_{14}
ROW_3	b_3	b_7	b_{11}	$S_{3,3}=b_{15}$
	COL_0	COL_1	COL_2	COL_3

Figure 1 The main encryption processing first moves the plain text to a 4×4-byte matrix called the "state."

$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$
$S_{1,0}$	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$
$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$
$S_{3,0}$	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$



$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$
$S_{1,1}$	$S_{1,2}$	$S_{1,3}$	$S_{1,0}$
$S_{2,2}$	$S_{2,3}$	$S_{2,0}$	$S_{2,1}$
$S_{3,3}$	$S_{3,0}$	$S_{3,1}$	$S_{3,2}$

Figure 2 In the ShiftRows() transformation, the column order circularly shifts a number of positions, dependent on the row number.

- *Cipher text* is the encrypted data.
- *Plain text* is the original unencrypted data.
- The AES algorithm expands the 128-, 192-, or 256-bit key into a number of 32-bit values. Each round receives a new 128-bit *key*. The total size of the key schedule depends on the key size.
- An *S-box*, or substitution box, is a look-up table.

EXPANDING INTO A KEY SCHEDULE

The AES algorithm expands the initial encryption key into a table of 32-bit values. The algorithm then subdivides the table into groups of four 32-bit values. Each round of the AES algorithm uses a key made up of four 32-bit values from the key schedule.

The total number of scheduled keys, and thus the total table size, depends on the initial key size. A scheduled key consists of four 32-bit values for each round of the algorithm and one final set of four 32-bit values. For instance, with a 128-bit key, there are 10 rounds, so the number of 32-bit values generated during the key expansion is 44. Likewise, the table size for a 192-bit key is 52 32-bit values long, and the size for a 256-bit key is 60 32-bit values long.

Listing 1 shows the algorithm for key expansion. Explanations of some of the terms in the pseudocode are as follows:

- the "key" is stored as an array of bytes and contains the encryption key;
- "key_size" is the size of the key array divided by four (that is, the number of 32-bit words that the key array forms);
- "w" is the final key-schedule array and is stored as a number of 32-bit values;
- "i" is an index variable;
- "temp" is a 32-bit temporary variable;
- "Nr" is the number of rounds, which depends on the key size;
- "word()" forms a word out of four bytes;
- "SubWord()" is a byte-by-byte substitution of a 32-bit word using the S-box look-up table; and
- "Rcon[i]" is a look-up-table value that the word formed from the byte-wide values, $\{2^{i-1}, 0, 0, 0\}$, gives. However, there is a slight trick to this calculation. The key-expansion algorithm calculates the first byte as an exponential within a "Galois field." Find out more about the Galois field by visiting the Web version of this article at www.edn.com and reading the Web-exclusive sidebar. The easiest way to implement this table is to use the example table values in the AES standard.

LISTING 1—KEY-EXPANSION ALGORITHM

```
KeyExpansion(key[], w[4+(Nr*4], key_size)
{

Copy the key to the first values of the key schedule
for (i=0; i < key_size; i++) {
            w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
            }

Expand the rest of the key schedule values
for(i=key_size; i < (4 +Nr*4); i++) {
            temp = w[i-1]
            if (i mod key_size = 0)
                        temp = SubWord(RotWord(temp)) xor
                        Rcon[ i / key_size ]
            else if ((key_size > 6) and (i mod key_size == 4))
                        temp = SubWord(temp)

            w[i] = w[ i - key_size ] xor temp
            }
}
```

LISTING 2—AES CIPHER ALGORITHM

```
Cipher(in[16], out[16], w[4+(Nr*4)])
{
byte state[4][4]

Copy the plaintext array to the matrix array
state = in

Add the round key for round 0
AddRoundKey(state, w, 0)

Begin rest of rounds
for (round = 1; round< Nr–1; round ++) {
            SubBytes(state)
            ShiftRows(state)
            MixColumns(state)
            AddRoundKey(state, w, round)
            }

Final processing
            SubBytes(state)
            ShiftRows(state)
            AddRoundKey(state, w, Nr)

Copy the state matrix to the ciphertext array
out = state
}
```

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix}$$

Figure 3

The MixColumns() matrix operation multiplies each column (in a Galois field) by a 4×4-byte matrix.

THE AES ENCRYPTION ALGORITHM

The AES algorithm converts 128 bits of plain text to 128 bits of cipher text. The main encryption processing first moves the plain text (best thought of as an array of 16 bytes) to a 4×4-byte matrix called the "state." **Figure 1** shows the transformation from the plain-text array into the state matrix. Further suboperations of the AES algorithm happen on the state matrix. When all processing is done, the final operation moves the state matrix to a cipher-text array with the same byte ordering that initial cipher operation used to move the plain-text array to the state matrix.

Listing 2 displays pseudocode for the AES cipher algorithm. As mentioned before, the number of rounds (Nr) depends on the key size. For a 128-bit key, Nr=10; for a 192-bit key, Nr=12; and for a 256-bit key, Nr=14. Note that the cipher contains four main suboperations: AddRoundKey(), SubBytes(), ShiftRows(), and MixColumns().

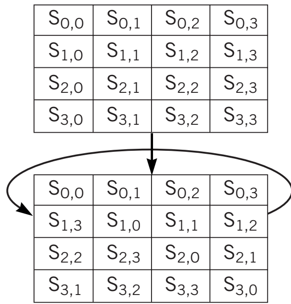
AddRoundKey() performs an exclusive-OR of each column with a 32-bit value from the key schedule for that round. As previously mentioned, the key schedule divides into groups of four 32-bit values, and, likewise, the state matrix has four columns. The column number c acts as an offset into the key schedule for the specified round to address the correct 32-bit key. For each column c,

$$\{S_{0,c}, S_{1,c}, S_{2,c}, S_{3,c}\} = \{S_{0,c}, S_{1,c}, S_{2,c}, S_{3,c}\} \text{ xor } w[round*4 + c]$$

SubBytes() substitutes each byte in the state matrix with a new value. It splits each byte in the state matrix into two 4-bit nibbles. SubBytes() uses each nibble to address a 2-D 16×16-byte look-up table, or S-box. The high nibble addresses the row, and the low nibble addresses the column of the S-box such that $S' = \text{S-box}[S_{high}][S_{low}]$.

ShiftRows() switches the order of the column entries for a single row in the state matrix. The ShiftRows() operation acts independently upon each row. Depending on the row number, the column order shifts circularly a number of positions. This

Figure 4



The InvShiftRows() transformation returns the inverse of the SubBytes() operation for each byte entry in the state matrix.

operation does not change row 0. Row 1 circularly shifts to the left by one column. Likewise, row 2 shifts to the left by 2, and row 3 shifts to the left by 3. **Figure 2** graphically depicts the operation.

MixColumns() mixes the values for any given column with other values from the same column. The MixColumns() operation is the most complex aspect of the AES algorithm. However, it is comprehensible when you twist your brain the right way. MixColumns() uses the concept of multiplying two numbers in a Galois field, $GF(2^8)$. Operating on a single column at a time, MixColumns() multiplies the column (in a Galois field) by a 4×4-byte matrix. **Figure 3** shows the operation.

The result of the multiplication in the **figure** is a set of four equations that can be quickly calculated using shifts and exclusive-OR operations to determine the new values for each column c in the state matrix:

$$S'_{0,c} = (\{02\} \bullet S_{0,c}) \text{ xor } (\{03\} \bullet S_{1,c}) \text{ xor } S_{2,c} \text{ xor } S_{3,c}$$
$$S'_{1,c} = S_{0,c} \text{ xor } (\{02\} \bullet S_{1,c}) \text{ xor } (\{03\} \bullet S_{2,c}) \text{ xor } S_{3,c}$$
$$S'_{2,c} = S_{0,c} \text{ xor } S_{1,c} \text{ xor } (\{02\} \bullet S_{2,c}) \text{ xor } (\{03\} \bullet S_{3,c})$$
$$S'_{3,c} = (\{03\} \bullet S_{0,c}) \text{ xor } S_{1,c} \text{ xor } S_{2,c} \text{ xor } (\{02\} \bullet S_{3,c})$$

THE AES INVERSE CIPHER

The inverse cipher converts the cipher text that the AES algorithm produces to the original plain text. The cipher and deciphering algorithms share the same key schedule but use the scheduled keys in opposite order.

Listing 3 contains pseudocode for the inverse-cipher algorithm. The deciphering algorithm starts and ends with the same moves between the I/O array and the state matrix that the cipher algorithm used. Also, the deciphering algorithm performs the same number of rounds as did the cipher algorithm.

LISTING 3–AES INVERSE CIPHER

```
InvCipher(in[16], out[16], word w[4* (Nr*4)])
{
Move the input array to the state matrix
state = in

Add the last round key
AddRoundKey(state, w, Nr)

Perform the deciphering rounds
for(round=Nr-1; round>0; round--) {
        InvShiftRows(state)
        InvSubBytes(state)
        AddRoundKey(state, w, round)
        InvMixColumns(state)
        }

Perform the final operations
InvShiftRows(state)
InvSubBytes(state)
AddRoundKey(state, w, 0)

Move the state matrix to the output array
out = state
}
```

Note that the new suboperations start with the prefix "Inv." These operations are simply the inverse of the cipher-algorithm suboperations.

InvShiftRows() simply moves the column entries in a single row back to where they were before the ShiftRows() operation applied its row shift. **Figure 4** shows a graphical representation of the reordering of the column entries in each row. As before, row 0 is unaffected. Row 1 circularly shifts one column to the right, row 2 shifts two positions, and row 3 shifts three positions.

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix}$$

Figure 5

The InvMixColumns() matrix operation uses a new 4×4-byte matrix to perform the multiplication.

The InvSubBytes() operation returns the inverse of the SubBytes() operation for each byte entry in the state matrix. The deciphering algorithm carries out InvSubBytes() in exactly the same manner as the SubBytes routine; however, it uses a new 16×16-byte look-up table. (See **Reference 1** for the inverse look-up table.)

InvMixColumns() is a little more complex to perform but requires no more knowledge than the MixColumns() operation. InvMixColumns() uses a new 4×4-byte matrix to perform the Galois-field multiplication. **Figure 5** shows the new multiplication, and the text that follows shows the resulting equations. Just like the MixColumns() operation, InvMixColumns() performs the multiplication on a single column at a time.

Although the larger multiplication factors are a bit more computation-intensive, the InvMixColumns() operation carries out the multiplication in the exact same manner as the MixColumns() multiplication:

$$S'_{0,c} = (\{0E\} \bullet S_{0,c}) \text{ xor } (\{0B\} \bullet S_{1,c}) \text{ xor } (\{0D\} \bullet S_{2,c}) \text{ xor } (\{09\} \bullet S_{3,c})$$
$$S'_{1,c} = (\{09\} \bullet S_{0,c}) \text{ xor } (\{0E\} \bullet S_{1,c}) \text{ xor } (\{0B\} \bullet S_{2,c}) \text{ xor } (\{0D\} \bullet S_{3,c})$$
$$S'_{2,c} = (\{0D\} \bullet S_{0,c}) \text{ xor } (\{09\} \bullet S_{1,c}) \text{ xor } (\{0E\} \bullet S_{2,c}) \text{ xor } (\{0B\} \bullet S_{3,c})$$
$$S'_{3,c} = (\{0B\} \bullet S_{0,c}) \text{ xor } (\{0D\} \bullet S_{1,c}) \text{ xor } (\{09\} \bullet S_{2,c}) \text{ xor } (\{0E\} \bullet S_{3,c})$$

APPLICATIONS OF THE AES

Because the NIST only recently approved the AES algorithm, it has not yet been deployed into as many applications as DES. The AES seems first to be making headway in secure-IP (Internet Protocol) networking. The latest version of the Unix OS NetBSD incorporates the AES algorithm for its implementation of the networking protocol IPSec. Because secure-networking protocols tend to use the most cryptographically strong cipher shared between two nodes, and network nodes tend to be software upgradeable, security services should fully deploy the AES in the short term.□

References

1. NIST AES standard document, FIPS-197, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.
2. AES algorithm press release, www.nist.gov/public_affairs/releases/g00-176.htm.

Author's biography

Stuart Allman is a senior systems engineer and solution architect for the Personal Communications Division of Cypress Semiconductor (www.cypress.com). He holds a BS degree in electrical engineering from the University of Washington (Seattle).

GALOIS FIELD

To understand a Galois field, consider a byte-wide bit field {b}. Each bit b_n of the byte b represents a power of x. For instance, you can represent b using the polynomial form below, where b_n represents the bits within b:

$$\{b\} = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0$$

Thus, for b=0×3A = 00111010,

$$\{3A\} = x^5 + x^4 + x^3 + x^1$$

Any addition or subtraction operation is performed via an exclusive-OR logical operation in this field. For instance,

$$\{3A\} + \{2B\} = 0x3A \text{ xor } 0x2B = 0x11$$

Likewise,

$$\{3A\} - \{2B\} = 0x3A \text{ xor } 0x2B = 0x11$$

Multiplication in the field is performed via polynomial multiplication, using the aforementioned transform of the byte b to powers of x. For instance,

$$\{3A\}\bullet\{2B\} = (x^5 + x^4 + x^3 + x^1)(x^5 + x^3 + x^1 + 1)$$
$$= x^{10} + x^8 + x^6 + x^5 + x^9 + x^7 + x^5 + x^4 + x^8 + x^6 + x^4 + x^3 + x^6 + x^4 + x^2 + x^1$$

Keep in mind that addition is performed in the field with an exclusive-OR operation, so the polynomial multiplication can be reduced to:

$$= x^{10} + x^9 + x^7 + x^6 + x^4 + x^3 + x^2 + x^1$$

An additional property is necessary to complete the multiplication operation. Any operation within the field results in a value also within the field. In this case, the result is confined to an 8-bit value, because the field is defined as $GF(2^8)$. To fulfill this property, any multiplication operation is performed modulo an irreducible polynomial that restricts the result to an 8-bit value. For the AES algorithm, this polynomial is

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

Therefore, the final result can be calculated with polynomial division. However, exclusive-OR operations replace intermediate subtractions. So,

$$= x^{10} + x^9 + x^7 + x^6 + x^4 + x^3 + x^2 + x^1 \text{ modulo } x^8 + x^4 + x^3 + x + 1$$
$$= x^7 + x^2 = \{14\}$$

There is a simpler way to calculate the result in software or hardware. Each time b is multiplied by {02} (also known as x), the value of b shifts to the left by 1 bit. If the 9-bit result has the most significant bit set, then the modulus operation is performed by simply subtracting m(x). As previously mentioned, a subtraction is simply a logical exclusive-OR.

If the most significant bit after the shift operation is not set, then the least significant 8 bits are already the result of the modulo operation. If you repeat the multiplication of the previous result and {02}, the result is equivalent to multiplying the original b by {04}, and so forth, until you are multiplying b by {0x80}. Now that all of the values of b have been multiplied by the powers of 2, the results are added (that is, exclusive-OR) together to form the final result. As an example, consider the following:

$$\{81\}\bullet\{05\} = \{81\} \bullet (\{04\} \text{ xor } \{01\})$$

Calculate b times each power of 2 to get:

$$\{81\}\bullet\{01\} = \{81\}$$
$$\{81\}\bullet\{02\} = \{19\}$$
$$\{81\}\bullet\{04\} = \{19\}\bullet\{02\} = \{32\}$$
$$\{81\}\bullet\{08\} = \{32\}\bullet\{02\} = \{64\}$$
$$\ldots$$

Calculate the final result from the {04} and {01} terms above to get:

$$= \{32\} \text{ xor } \{81\} = \{B3\}$$