Lecture 8: AES: The Advanced Encryption Standard

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

February 26, 2013

2:21pm

Goals:

- To review the overall structure of AES.

- To focus particularly on the four steps used in each round of AES: (1) substitution, (2) shift rows, (3) mix columns, and (4) add round key.

- To go through the details of how the encryption key is expanded to yield the round keys.

CONTENTS

8.1: SALIENT FEATURES OF AES

- AES is a block cipher with a block length of 128 bits.

- AES allows for three different key lengths: 128, 192, or 256 bits. Most of our discussion will assume that the key length is 128 bits. [With regard to using a key length other than 128 bits, the main thing that changes in AES is how you generate the *key schedule* from the key — an issue I address at the end of Section 8.8.1. The notion of *key schedule* in AES is explained in Sections 8.2 and 8.8.]

- Encryption consists of 10 rounds of processing for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys.

- Except for the last round in each case, all other rounds are identical.

- Each round of processing includes one single-byte based substitution step, a row-wise permutation step, a column-wise mixing step, and the addition of the round key. The order in which these four steps are executed is different for encryption and decryption.

- To appreciate the processing steps used in a single round, it is best to think of a 128-bit block as consisting of a 4×4 matrix of bytes, arranged as follows:

$$\begin{bmatrix} byte_0 & byte_4 & byte_8 & byte_{12} \\ byte_1 & byte_5 & byte_9 & byte_{13} \\ byte_2 & byte_6 & byte_{10} & byte_{14} \\ byte_3 & byte_7 & byte_{11} & byte_{15} \end{bmatrix}$$

- Therefore, the first four bytes of a 128-bit input block occupy the first column in the 4×4 matrix of bytes. The next four bytes occupy the second column, and so on.

- The 4×4 matrix of bytes is referred to as the **state array**.

- AES also has the notion of a **word**. A word consists of four bytes, that is 32 bits. Therefore, each column of the state array is a word, as is each row.

- Each round of processing works on the **input state array** and produces an **output state array**.

- The output state array produced by the last round is rearranged into a 128-bit output block.

- Unlike DES, the decryption algorithm differs substantially from the encryption algorithm. Although, overall, the same steps are used in encryption and decryption, the order in which the steps are carried out is different, as mentioned previously.

- AES, notified by NIST as a standard in 2001, is a slight variation of the Rijndael cipher invented by two Belgian cryptographers Joan Daemen and Vincent Rijmen.

- Whereas AES requires the block size to be 128 bits, the original Rijndael cipher works with any block size (and any key size) that is a multiple of 32 as long as it exceeds 128. The state array for the different block sizes still has only four rows in the Rijndael cipher. However, the number of columns depends on size of the block. For example, when the block size is 192, the Rijndael cipher requires a state array to consist of 4 rows and 6 columns.

• As explained in Lecture 3, DES was based on the Feistel network. On the other hand, what AES uses is a substitution-permutation network in a more general sense. Each round of processing in AES involves byte-level substitutions followed by word-level permutations. Speaking generally, DES also involves substitutions and permutations, except that the permutations are based on the Feistel notion of dividing the input block into two halves, processing each half separately, and then swapping the two halves. The nature of substitutions and permutations in AES allows for a fast software implementation of the algorithm.

• The AES standard is described in the following official document:

  http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

8.2: THE ENCRYPTION KEY AND ITS EXPANSION

- Assuming a 128-bit key, the key is also arranged in the form of a matrix of 4×4 bytes. As with the input block, the first word from the key fills the first column of the matrix, and so on.

- The four column words of the key matrix are expanded into a schedule of 44 words. (As to how exactly this is done, we will explain that later.) Each round consumes four words from the key schedule.

- The figure below depicts the arrangement of the encryption key in the form of 4-byte words and the expansion of the key into a **key schedule** consisting of 44 4-byte words.
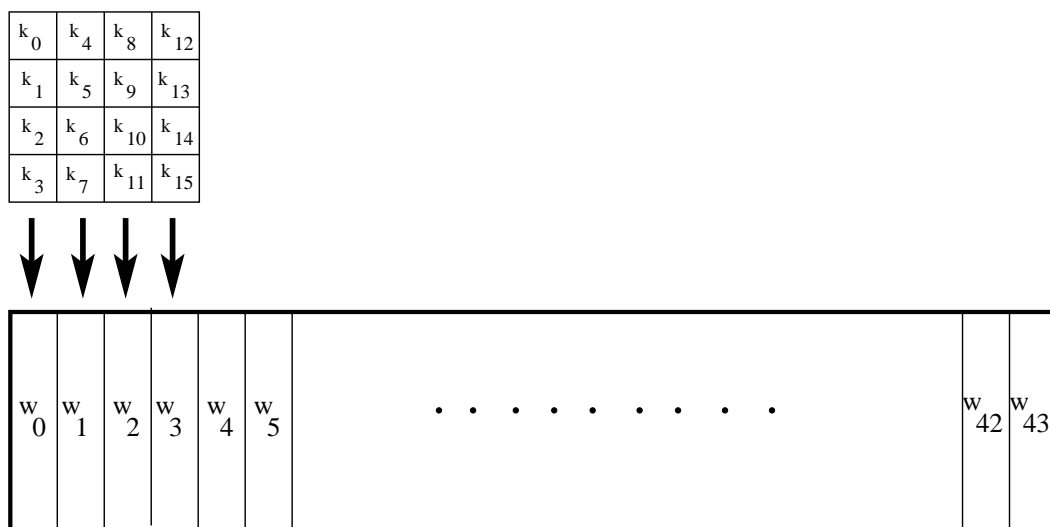
Figure 1: *This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak*

8.3: THE OVERALL STRUCTURE OF AES

- The overall structure of AES encryption/decryption is shown in Figure 2.

- The number of rounds shown in Figure 2, 10, is for the case when the encryption key is 128 bit long. (As mentioned earlier, the number of rounds is 12 when the key is 192 bits, and 14 when the key is 256.)

- Before any round-based processing for encryption can begin, the input state array is XORed with the first four words of the key schedule. The same thing happens during decryption — except that now we XOR the ciphertext state array with the last four words of the key schedule.

- For encryption, each round consists of the following four steps: 1) Substitute bytes, 2) Shift rows, 3) Mix columns, and 4) Add round key. The last step consists of XORing the output of the previous three steps with four words from the key schedule.

- For decryption, each round consists of the following four steps: 1) Inverse shift rows, 2) Inverse substitute bytes, 3) Add round key, and 4) Inverse mix columns. The third step consists of XORing the output of the previous two steps with four words from the key schedule. Note the differences between the order in which substitution and shifting operations are carried out in a decryption round vis-a-vis the order in which similar operations are carried out in an encryption round.

- The last round for encryption does not involve the "Mix columns" step. The last round for decryption does not involve the "Inverse mix columns" step.

128 bit plaintext block

128 bit plaintext block

Add round key		Round 10

$w_0 - w_3$ $w_0 - w_3$

Round 1		Round 9

$w_4 - w_7$ $w_4 - w_7$

Round 2		Round 8

$w_8 - w_{11}$ $w_8 - w_{11}$

Key Schedule

Round 10		Add round key

$w_{40} - w_{43}$ $w_{40} - w_{43}$

128 bit ciphertext block

128 bit ciphertext block

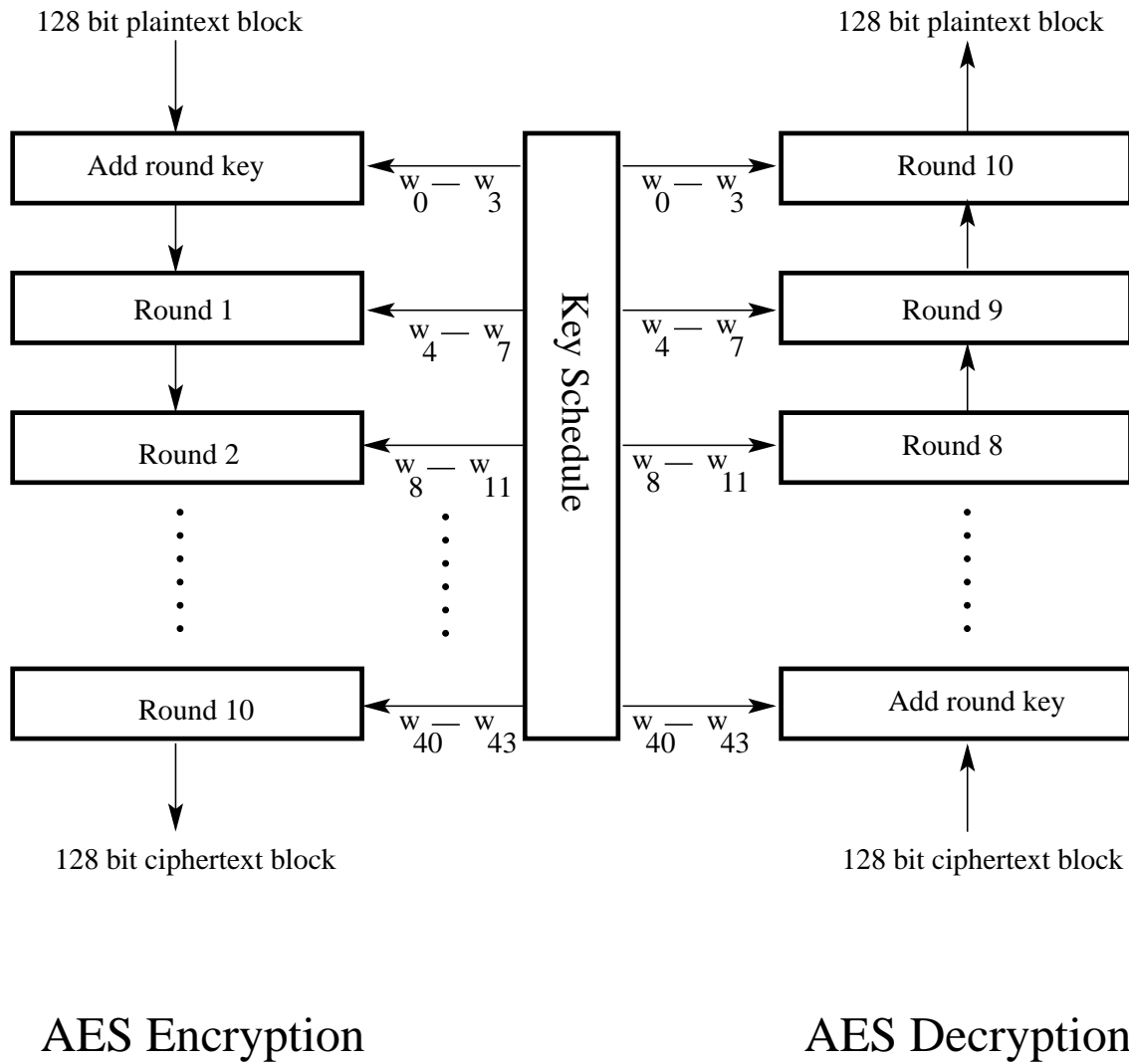## AES Encryption   AES Decryption

Figure 2: *This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak*

8.4: THE FOUR STEPS IN EACH ROUND OF PROCESSING

Figure 3 shows the different steps that are carried out in each round except the last one.

STEP 1: (called **SubBytes** for byte-by-byte substitution during the forward process) (The corresponding substitution step used during decryption is called **InvSubBytes**.)

- This step consists of using a 16×16 lookup table to find a replacement byte for a given byte in the input state array.

- The entries in the lookup table are created by using the notions of multiplicative inverses in $GF(2^8)$ and bit scrambling to destroy the bit-level correlations inside each byte.   [See Lecture 7 for what is meant by the notation $GF(2^8)$.]

STEP 2: (called **ShiftRows** for shifting the rows of the state array during the forward process) (The corresponding transformation during decryption is denoted **InvShiftRows** for Inverse Shift-Row Transformation.)

Substitute Bytes

Shift Rows

Mix Columns

Add Round Key ← Round Key

Encryption Round

Inverse Mix Columns

Round Key → Add Round Key

Inverse Substitute Bytes

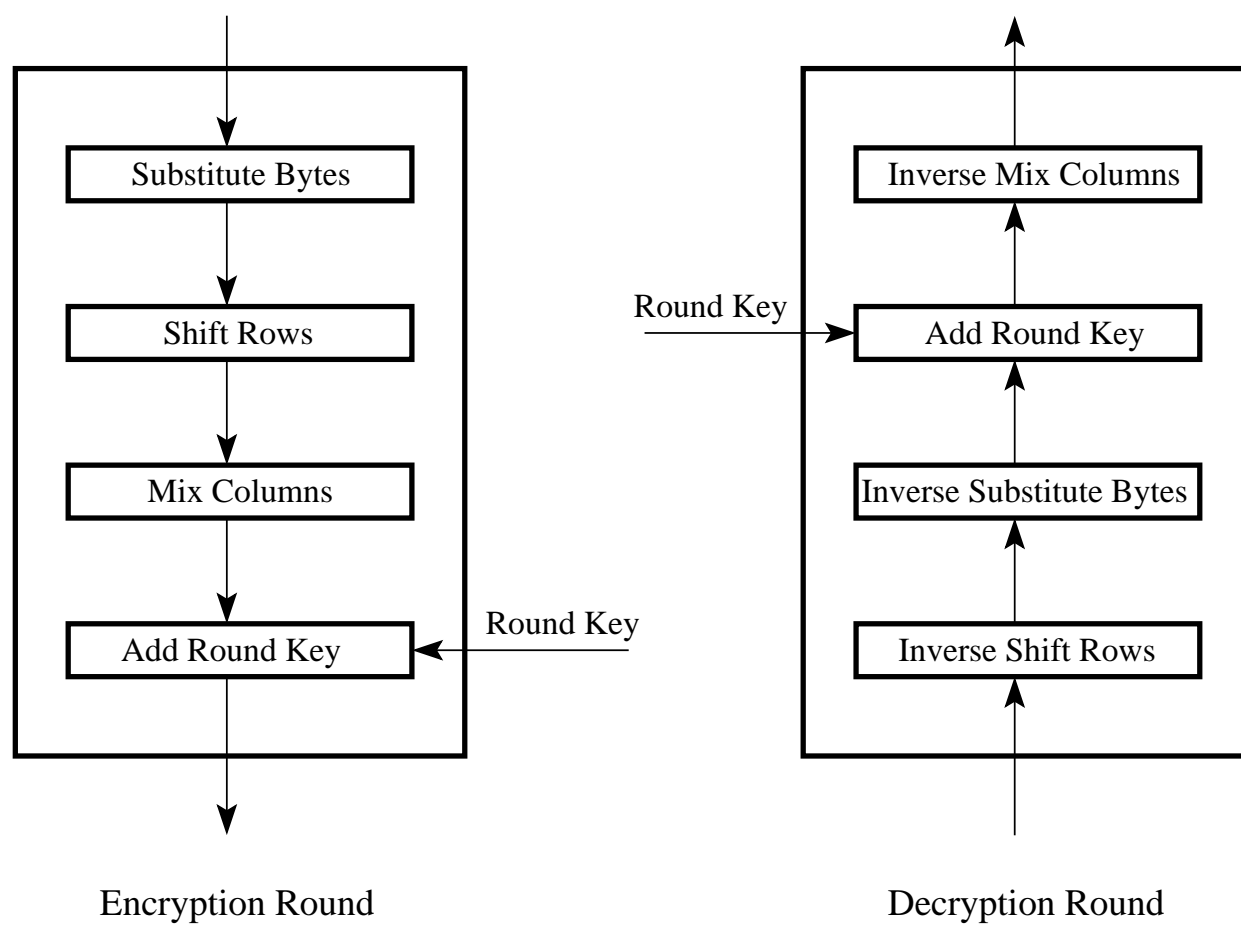Inverse Shift Rows

Decryption Round

Figure 3: *This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak*

• The goal of this transformation is to scramble the byte order inside each 128-bit block.

STEP 3: (called **MixColumns** for mixing up of the bytes in each column separately during the forward process) (The corresponding transformation during decryption is denoted **InvMixColumns** and stands for inverse mix column transformation.) The goal is here is to further scramble up the 128-bit input block.

  • The shift-rows step along with the mix-column step causes each bit of the ciphertext to depend on every bit of the plaintext after 10 rounds of processing.

  • Recall the avalanche effect from our discussion on DES in Lecture 3. In DES, one bit of plaintext affected roughly 31 bits of ciphertext. But now we want each bit of the plaintext to affect every bit of the ciphertext in a block of 128 bits.

STEP 4: (called **AddRoundKey** for adding the round key to the output of the previous step during the forward process) (The corresponding step during decryption is denoted **InvAddRound-Key** for inverse add round key transformation.)

8.5: THE SUBSTITUTE BYTES STEP (SubBytes)

- This is a byte-by-byte substitution. The substitution byte for each input byte is found by using the same lookup table.

- The size of the lookup table is 16×16.

- To find the substitute byte for a given input byte, we divide the input byte into two 4-bit patterns, each yielding an integer value between 0 and 15. (We can represent these by their hex values 0 through F.) One of the hex values is used as a row index and the other as a column index for reaching into the 16×16 lookup table.

- The entries in the lookup table are constructed by a combination of $GF(2^8)$ arithmetic and bit mangling.

- The goal of the substitution step is to reduce the correlation between input bits and output bits (at the byte level). The bit mangling part of the substitution step ensures that the substi-

tution cannot be described in the form of evaluating a simple mathematical function.

8.5.1: Construction of the 16×16 Lookup Table for the SubBytes Step

- We first fill each cell of the 16×16 table with the byte obtained by joining together its row index and the column index.

- For example, for the cell located at row index 2 and column indexed 7, we place 27 in the cell. So at this point the table will look like

```
          0     1     2     3     4     5     6     7     8     9 ....
        ---------------------------------------------------------
  0   |  00    01    02    03    04    05    06    07    08    09  ....
      |
  1   |  10    11    12    13    14    15    16    17    18    19  ....
      |
  2   |  20    21    22    23    24    25    26    27    28    29  ....
      |
           . . . . . . . . .
           . . . . . . . . .
```

- We next replace the value in each cell by its multiplicative inverse in $GF(2^8)$ based on the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$. **The value 00 is replaced by itself since this element**

has no multiplicative inverse. [See Lecture 7 for what we mean by the multiplicative inverse of a byte modulo an irreducible polynomial and as to why the zero byte has no multiplicative inverse.]

- After the above steop, let's represent a byte stored in a cell of the table by $b_7b_6b_5b_4b_3b_2b_1b_0$ where b_7 is the MSB and b_0 the LSB. For example, the byte stored in the cell $(9, 5)$ of the above table is the **multiplicative inverse** (MI) of $0x95$, which is $0x8A$. Therefore, at this point, the bit pattern stored in the cell with row index 9 and column index 5 is 10001010, implying that b_7 is 1 and b_0 is 0. [Verify the fact that the MI of $0x95$ is indeed $0x8A$. The polynomial representation of $0x95$ (bit pattern: 10010101) is $x^7 + x^4 + x^2 + 1$, and the same for $0x8A$ (bit pattern: 10001010) is $x^7 + x^3 + x$. Now show that the product of these two polynomials modulo the polynomial $x^8 + x^4 + x^3 + x + 1$ is indeed 1.]

- For bit mangling, we next apply the following transformation to each **bit** b_i of the byte stored in a cell of the lookup table:

$$b_i' = b_i \otimes b_{(i+4) \ mod \ 8} \otimes b_{(i+5) \ mod \ 8} \otimes b_{(i+6) \ mod \ 8} \otimes b_{(i+7) \ mod \ 8} \otimes c_i$$

where c_i is the ith bit of a specially designated byte c whose hex value is $Ox63$. ($c_7c_6c_5c_4c_3c_2c_1c_0 = 01100011$)

- The above bit-mangling step can be better visualized as the following vector-matrix operation. Note that all of the additions in the product of the matrix and the vector are actually XOR operations. [Because of the $[A]\vec{x} + \vec{b}$ nature of this transformation, it is commonly referred to as the affine transformation.]

$$
\begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1
\end{bmatrix}
\begin{bmatrix}
b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7
\end{bmatrix}
\otimes
\begin{bmatrix}
1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0
\end{bmatrix}
$$

- The 16×16 table created in this manner is called the **S-Box**. The **S-Box** is the same for all the bytes in the **state array**.

- The extra bit-mangling introduced by the transformation shown above is meant to break the correlation between the bits before the substitution and the bits after the substitution while maintaining a relationship between the input bytes and the output bytes that can be described by algebraic equations.

- This byte-by-byte substitution step is reversed during decryption,

meaning that you first apply the reverse of the bit-mangling operation to each byte, as explained in the next step, and then you take its multiplicative inverse in $GF(2^8)$.

- The 16×16 lookup table for decryption is constructed by starting out in the same manner as for the encryption lookup table. That is, you place in each cell the byte constructed by joining the row index with the column index. Then, for bit mangling, you carry out the following bit-level transformation in each cell of the table:

$$b'_i \;\; = \;\; b_{(i+2) \mod 8} \otimes b_{(i+5) \mod 8} \otimes b_{(i+7) \mod 8} \otimes d_i$$

where d_i is the ith bit of a specially designated byte d whose hex value is $Ox05$. ($d_7 d_6 d_5 d_4 d_3 d_2 d_1 d d c_0 \;\; = \;\; 00000101$) Finally, you replace the byte in the cell by its multiplicative inverse in $GF(2^8)$.

- The bytes c and d are chosen so that the S-box has no fixed points. That is, we do not want $S\_box(a) \;\; = \;\; a$. Neither do we want $S\_box(a) \;\; = \;\; \bar{a}$ where $\bar{a}$ is the bitwise complement of a.

8.6: THE SHIFT ROWS STEP (ShiftRows)

• This is where the matrix representation of the **state array** becomes important.

• The ShiftRows transformation consists of (i) not shifting the first row of the **state array** at all; (ii) circularly shifting the second row by one byte to the left; (iii) circularly shifting the third row by two bytes to the left; and (iv) circularly shifting the last row by three bytes to the left.

• This operation on the state array can be represented by

$$
\begin{bmatrix}
s_{0.0} & s_{0,1} & s_{0,2} & s_{0,3} \\
s_{1.0} & s_{1,1} & s_{1,2} & s_{1,3} \\
s_{2.0} & s_{2,1} & s_{2,2} & s_{2,3} \\
s_{3.0} & s_{3,1} & s_{3,2} & s_{3,3}
\end{bmatrix}
\; ===> \;
\begin{bmatrix}
s_{0.0} & s_{0,1} & s_{0,2} & s_{0,3} \\
s_{1.1} & s_{1,2} & s_{1,3} & s_{1,0} \\
s_{2.2} & s_{2,3} & s_{2,0} & s_{2,1} \\
s_{3.3} & s_{3,0} & s_{3,1} & s_{3,2}
\end{bmatrix}
$$

• Recall again that the input block is written column-wise. That is the first four bytes of the input block fill the first column of

the state array, the next four bytes the second column, etc. As a result, shifting the rows in the manner indicated scrambles up the byte order of the input block.

- For decryption, the corresponding step shifts the rows in exactly the opposite fashion. The first row is left unchanged, the second row is shifted to the right by one byte, the third row to the right by two bytes, and the last row to the right by three bytes, all shifts being circular.

$$
\begin{bmatrix}
s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\
s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\
s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\
s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3}
\end{bmatrix}
===>
\begin{bmatrix}
s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\
s_{1,3} & s_{1,0} & s_{1,1} & s_{1,2} \\
s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\
s_{3,1} & s_{3,2} & s_{3,3} & s_{3,0}
\end{bmatrix}
$$

8.7: THE MIX COLUMNS STEP (MixColumns)

- This step replaces each byte of a column by a function of all the bytes in the same column.

- More precisely, each byte in a column is replaced by two times that byte, plus three times the the next byte, plus the byte that comes next, plus the byte that follows. The words 'next' and 'follow' refer to bytes in the same column, and their meaning is circular, in the sense that the byte that is next to the one in the last row is the one in the first row. [By 'two times' and 'three times', we mean multiplications in $GF(2^8)$ by the bit patterns 000000010 and 00000011, respectively.]

- For the bytes in the **first row** of the state array, this operation can be stated as

$$s'_{0,j} \;\;=\;\; \left(2 \times s_{0,j}\right) \;\otimes\; \left(3 \times s_{1,j}\right) \;\otimes\; s_{2,j} \;\otimes\; s_{3,j}$$

- For the bytes in the **second row** of the state array, this operation can be stated as

$$s'_{1,j} \;\; = \;\; s_{0,j} \;\; \otimes \;\; \left(2 \times s_{1,j}\right) \;\; \otimes \;\; \left(3 \times s_{2,j}\right) \;\; \otimes \;\; s_{3,j}$$

- For the bytes in the **third row** of the state array, this operation can be stated as

$$s'_{2,j} \;\; = \;\; s_{0,j} \;\; \otimes \;\; s_{1,j} \;\; \otimes \;\; \left(2 \times s_{2,j}\right) \;\; \otimes \;\; \left(3 \times s_{3,j}\right)$$

- And, for the bytes in the **fourth row** of the state array, this operation can be stated as

$$s'_{3,j} \;\; = \;\; \left(3 \times s_{0,j}\right) \;\; \otimes \;\; s_{1,j} \;\; \otimes \;\; s_{2,j} \;\; \otimes \;\; \left(2 \times s_{3,j}\right)$$

- More compactly, the column operations can be shown as

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} s_{0.0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1.0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2.0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3.0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0.0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1.0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2.0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3.0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

where, on the left hand side, when a row of the leftmost matrix multiples a column of the state array matrix, additions involved are meant to be XOR operations.

- The corresponding transformation during decryption is given by

$$
\begin{bmatrix}
0E & 0B & 0D & 09 \\
09 & 0E & 0B & 0D \\
0D & 09 & 0E & 0B \\
0B & 0D & 09 & 0E
\end{bmatrix}
\times
\begin{bmatrix}
s_{0.0} & s_{0,1} & s_{0,2} & s_{0,3} \\
s_{1.0} & s_{1,1} & s_{1,2} & s_{1,3} \\
s_{2.0} & s_{2,1} & s_{2,2} & s_{2,3} \\
s_{3.0} & s_{3,1} & s_{3,2} & s_{3,3}
\end{bmatrix}
=
\begin{bmatrix}
s'_{0.0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\
s'_{1.0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\
s'_{2.0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\
s'_{3.0} & s'_{3,1} & s'_{3,2} & s'_{3,3}
\end{bmatrix}
$$

8.8: THE KEY EXPANSION ALGORITHM

- The 128 bits of the state array are bitwise XOR'ed with the 128 bits of the round key.

- The **AES Key Expansion** algorithm is used to derive the 128-bit round key from the original 128-bit encryption key.

- In the same manner as the 128-bit input block is arranged in the form of a state array, the algorithm first arranges the 16 bytes of the encryption key in the form of a 4×4 array of bytes

$$\begin{bmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} w_0 & w_1 & w_2 & w_3 \end{bmatrix}$$

- The first four bytes of the encryption key constitute the word w_0, the next four bytes the word w_1, and so on.

- The algorithm subsequently expands the words $[w_0, w_1, w_2, w_3]$ into a 44-word **key schedule** that can be labeled

$$w_0, \ w_1, \ w_2, \ w_3, \dots\dots\dots\dots, \ w_{43}$$

- Of these, the words $[w_0, w_1, w_2, w_3]$ are bitwise XOR'ed with the input block **before** the round-based processing begins.

- The remaining 40 words of the key schedule are used **four words at a time** in each of the 10 rounds.

- The above two statements are also true for decryption, except for the fact that we now reverse the order of the words in the key schedule, as shown in Figure 2: The last four words of the key schedule are bitwise XOR'ed with the 128-bit ciphertext block before any round-based processing begins. Subsequently, each of the four words in the remaining 40 words of the key schedule are used in each of the ten rounds of processing.

- Now comes the difficult part: How does the **Key Expansion Algorithm** expand four words w_0, w_1, w_2, w_3 into the 44 words $w_0, w_1, w_2, w_3, w_4, w_5, \ldots\ldots, w_{43}$?

- The key expansion algorithm will be explained in the next subsection with the help of Figure 4.

- But first note that the key expansion takes place on a four-word to four-word basis, in the sense that each grouping of four words decides what the next grouping of four words will be.
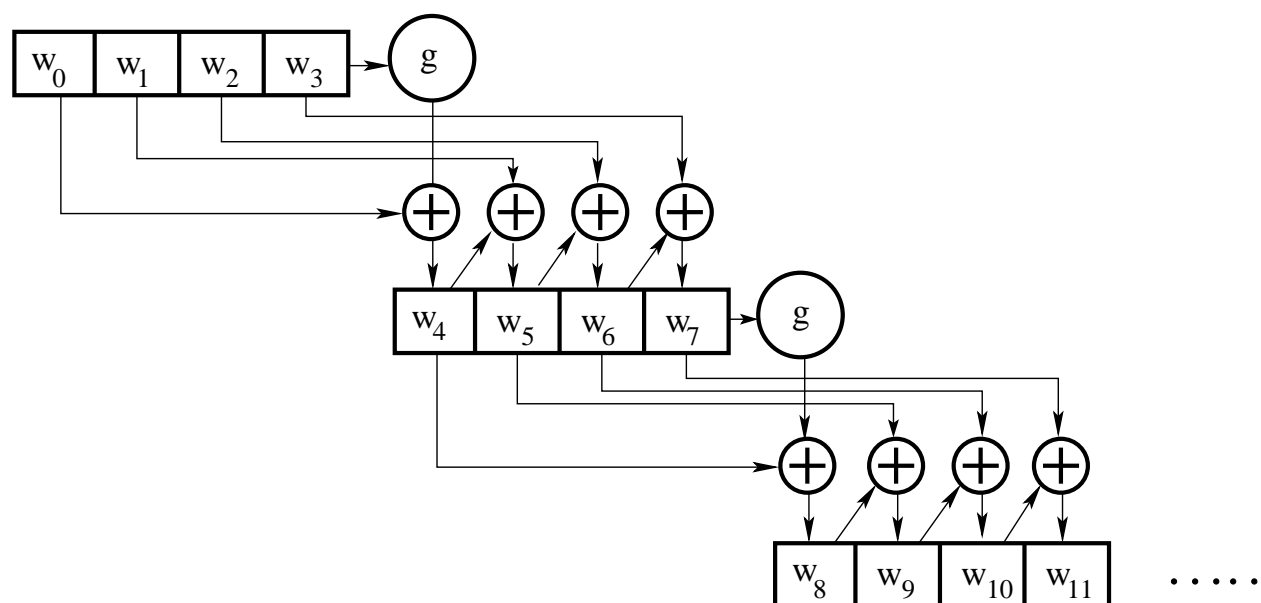
Figure 4: *This figure is from Lecture 8 of "Computer and Network Security"*

8.8.1: The Algorithmic Steps in Going from a 4-Word Round Key to the Next 4-Word Round Key

- We now come to the heart of the key expansion algorithm we talked about in the previous section — generating the four words of the round key for a given round from the corresponding four words of the round key for the previous round.

- Let's say that we have the four words of the round key for the ith round:

$$w_i \ w_{i+1} \ w_{i+2} \ w_{i+3}$$

For these to serve as the round key for the ith round, i must be a multiple of 4. These will obviously serve as the round key for the $(i/4)^{th}$ round. For example, w_4, w_5, w_6, w_7 is the round key for round 1, the sequence of words w_8, w_9, w_{10}, w_{11} the round key for round 2, and so on.

- Now we need to determine the words

$$w_{i+4} \ \ w_{i+5} \ \ w_{i+6} \ \ w_{i+7}$$

from the words $w_i \ \ w_{i+1} \ \ w_{i+2} \ \ w_{i+3}$.

- From Figure 4, we write

$$w_{i+5} \quad = \quad w_{i+4} \quad \otimes \quad w_{i+1} \tag{1}$$

$$w_{i+6} \quad = \quad w_{i+5} \quad \otimes \quad w_{i+2} \tag{2}$$

$$w_{i+7} \quad = \quad w_{i+6} \quad \otimes \quad w_{i+3} \tag{3}$$

Note that except for the first word in a new 4-word grouping, each word is an XOR of the previous word and the corresponding word in the previous 4-word grouping.

- So now we only need to figure out w_{i+4}. This is the beginning word of each 4-word grouping in the key expansion. The beginning word of each round key is obtained by:

$$w_{i+4} \quad = \quad w_i \otimes g(w_{i+3}) \tag{4}$$

That is, the first word of the new 4-word grouping is to be obtained by XOR'ing the first word of the last grouping with what is returned by applying a function $g()$ to the last word of the previous 4-word grouping.

- The function $g()$ consists of the following three steps:

– Perform a one-byte left circular rotation on the argument 4-byte word.

– Perform a byte substitution for each byte of the word returned by the previous step by using the same 16×16 lookup table as used in the **SubBytes** step of the encryption rounds.

– XOR the bytes obtained from the previous step with what is known as a **round constant**. The **round constant** is a word whose three rightmost bytes are always zero. Therefore, XOR'ing with the round constant amounts to XOR'ing with just its leftmost byte.

• The **round constant** for the ith round is denoted $Rcon[i]$. Since, by specification, the three rightmost bytes of the round constant are zero, we can write it as shown below. The left hand side of the equation below stands for the round constant to be used in the ith round. The right hand side of the equation says that the rightmost three bytes of the round constant are zero.

$$Rcon[i] \quad = \quad (RC[i], 0, 0, 0)$$

• The only non-zero byte in the round constants, $RC[i]$, obeys the following recursion:

$$
\begin{aligned}
RC[1] \quad &= \quad 1 \\
RC[j] \quad &= \quad 2 \times RC[j-1]
\end{aligned}
$$

- The addition of the round constants is for the purpose of destroying any symmetries that may have been introduced by the other steps in the key expansion algorithm.

- Our presentation of the key expansion algorithm is based on the assumption of a 128 bit key. As was mentioned in Section 8.1, AES calls for a larger number of rounds in Figure 2 when you use a key length other than 128 bits. A key length of 192 bits entails 12 rounds and a key length of 256 bits entails 14 rounds. (The length of the input block remains unchanged at 128 bits.) The key expansion algorithm must obviously generate a longer schedule for the 12 rounds required by a 192 bit key and the 14 rounds required by a 256 bit keys. Keeping in mind how we used the key schedule for the case of a 128 bit key, we are going to need 52 words in the key schedule for the case of 192-bit keys and 60 words for the case of 256-bit keys — with round-based processing remaining the same as described in Section 8.4. [Consider what happens when the key length is 192 bits. Since the round-based processing and the size of the input block remain the same as described earlier in this lecture, each round will still use only 4 words of the key schedule. Just as we organized the 128-bit key in the form of 4 key words for the purpose of key expansion, we organize the 192 bit key in the form of **six** words. The key expansion algorithm will take us from six

words to six words — for a total of **nine** key-expansion steps — with each step looking the same as what we described at the beginning of this section. Yes, it is true that the key expansion will now generate a total of 54 words while we need only 52 — we simply ignore the last two words of the key schedule. With regard to the details of going from the six words of the jth key-expansion step to the six words of the $(j+1)^{th}$ key expansion step, let's focus on going from the initial $(w_0, w_1, w_2, w_3, w_4, w_5)$ to $(w_6, w_7, w_8, w_9, w_{10}, w_{11})$. We generate the last five words of the latter from the last five words of the former through straightforward XORing as was the case earlier in this section. As for the first word of the latter, we generate it from the first and the last words of the former through the g function again as described earlier. The g function itself remains unchanged.] The cool thing about 128-bit keys is that you can think of the key expansion being in one-one correspondence with the rounds. However, that is no longer the case with, say, the 192-bit keys. Now you have to think of key expansion as something that is divorced even conceptually from round-based processing of the input block.

- Note that if you change one bit of the encryption key, it will affect the round key for several rounds.

- The key expansion algorithm ensures that AES has no **weak keys**. A weak key is a key that reduces the security of a cipher in a predictable manner. For example, DES is known to have weak keys. Weak keys of DES are those that produce identical round keys for each of the 16 rounds. An example of DES weak key is when it consists of alternating ones and zeros. This sort of a weak key in DES causes all the round keys to become identical,

which, in turn, causes the encryption to become self-inverting. That is, plain text encrypted and then encrypted again will lead back to the same plain text. (Since the small number of weak keys of DES are easily recognized, it is not considered to be a problem with that cipher.)

8.9: HOMEWORK PROBLEMS

1. AES is a block cipher. What sized blocks are used by AES?

2. How many rounds are used in AES and what does the number of rounds depend on?

3. What are the four steps that are executed in a single round of processing?

4. What is a state array in AES? Why do we need the notion of a state array in AES?

5. AES also uses the notion of a "word"? What is a word in AES? How many bits and bytes in a word?

6. In DES, the encryption and decryption algorithms are the same. Is that also the case in AES?

7. What is a key schedule in AES? How many words in a key schedule?

8. With regard to the first step of processing in each round of AES on the encryption side: How does one look up the 16×16 S-box table for byte-by-byte substitutions? In other words, assuming I want a substitute byte for the byte $b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$, where each b_i is a single bit, how do I use these bits to find the replacement byte in the S-box table?

9. What are the steps that go into the construction of the 16×16 S-box lookup table?

10. What is rationale for the bit scrambling step that is used for finding the replacement byte that goes into each cell of the S-box table?

11. The second step in each round permutes the bytes in each row of the state array. What is the permutation formula that is used?

12. Describe the "mix columns" transformation that constitutes the third step in each round of AES.

13. Let's now talk about the Key Expansion Algorithm of AES. This

algorithm starts with how many words of the key matrix and expands that into how many words?

14. Let's say the first four words of the key schedule are w_0, w_1, w_2, w_3. How do we now obtain the next four words w_4, w_5, w_6, w_7?

15. So the formula that yields w_4 is

$$w_4 \quad = \quad w_0 \otimes g(w_3)$$

What goes into computing $g()$?

16. **Programming Assignment:**

Write a Perl or Python script that implements one round of AES for encryption and for decryption. Your implementation must include the four steps that go into each round:

- byte-by-byte substitution
- shifting of the rows of the state array
- mixing of the columns
- the addition of the round key.

You should use a 128 bit encryption key. Your script should read a message file for the plaintext and write out the ciphertext into another file. It should prompt the user for the encryption key which should consist of at least 16 printable ASCII characters.

Acknowledgment

Thanks go to Parul Schroff (pschroff at purdue dot edu) for catching an error in Section 8.7.