Lecture 8: AES: The Advanced Encryption Standard

Lecture Notes on “Computer and Network Security”

by Avi Kak (kak@purdue.edu)

May 1, 2015
12:14 Noon

©2015 Avinash Kak, Purdue University

Goals:

• To review the overall structure of AES and to focus particularly on the four steps used in each round of AES: (1) byte 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.

• Python and Perl implementations for creating the lookup tables for the byte substitution steps in encryption and decryption.

• Perl implementations for creating histograms of the differentials and for constructing linear approximation tables in attacks on block ciphers.
## CONTENTS

<table>
<thead>
<tr>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Salient Features of AES</td>
<td>3</td>
</tr>
<tr>
<td>8.2 The Encryption Key and Its Expansion</td>
<td>8</td>
</tr>
<tr>
<td>8.3 The Overall Structure of AES</td>
<td>10</td>
</tr>
<tr>
<td>8.4 The Four Steps in Each Round of Processing</td>
<td>13</td>
</tr>
<tr>
<td>8.5 The Substitution Bytes Step: SubBytes and InvSubBytes</td>
<td>17</td>
</tr>
<tr>
<td>8.5.1 Constructing the $16 \times 16$ Lookup Tables for SubBytes and InvSubBytes</td>
<td>19</td>
</tr>
<tr>
<td>8.6 The Shift Rows Step: ShiftRows and InvShiftRows</td>
<td>27</td>
</tr>
<tr>
<td>8.7 The Mix Columns Step: MixColumns and InvMixColumns</td>
<td>29</td>
</tr>
<tr>
<td>8.8 The Key Expansion Algorithm</td>
<td>32</td>
</tr>
<tr>
<td>8.8.1 The Algorithmic Steps in Going from one 4-Word Round Key to the Next 4-Word Round Key</td>
<td>36</td>
</tr>
<tr>
<td>8.9 Differential, Linear, and Interpolation Attacks on Block Ciphers</td>
<td>42</td>
</tr>
<tr>
<td>8.10 Homework Problems</td>
<td>76</td>
</tr>
</tbody>
</table>
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 $4 \times 4$ matrix of bytes, arranged as follows:

$$
\begin{bmatrix}
\text{byte}_0 & \text{byte}_4 & \text{byte}_8 & \text{byte}_{12} \\
\text{byte}_1 & \text{byte}_5 & \text{byte}_9 & \text{byte}_{13} \\
\text{byte}_2 & \text{byte}_6 & \text{byte}_{10} & \text{byte}_{14} \\
\text{byte}_3 & \text{byte}_7 & \text{byte}_{11} & \text{byte}_{15}
\end{bmatrix}
$$

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

• The $4 \times 4$ matrix of bytes shown above is referred to as the state array in AES. [If you are trying to create your own implementation of AES in Python, you will find following statement, which uses the notion of list comprehension in Python, very useful for creating an initialized structure that looks like the state array of AES:
statearray = [[0 for x in range(4)] for x in range(4)]

Next, try the following calls in relation to the structure thus created:

```
print statearray
print statearray[0]
print statearray[2][3]
```

```
block = range(128)
```

```
for i in range(4):
    for j in range(4):
        statearray[j][i] = block[32*i + 8*j:32*i + 8*(j+1)]
```

```
for i in range(4):
    for j in range(4):
        print statearray[i][j], " ",
```

This is a nice warm-up exercise before you start implementing AES in Python.

- 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. [Back in 1999, the Rijndael cipher was one of the five selected by NIST as potential replacements for DES. The other four were: MARS from IBM; RC6 from RSA Security; Serpent by Ross Anderson, Eli Biham, and Lars Knudsen; and Twofish by a team led by the celebrated cryptographer Bruce Schneier. Rijndael was selected from amongst these five after extensive testing that was open to public.]

• 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:

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 $4 \times 4$ 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 in Section 8.8.) Each round consumes four words from the key schedule.

- Figure 1 on the next page 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 shows the four words of the original 128-bit key being expanded into a key schedule consisting of 44 words. Section 8.8 explains the procedure used for this key expansion. (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.
Figure 2: The overall structure of AES for the case of 128-bit encryption key. (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 $16 \times 16$ 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 $GF(2^8)$.] Section 8.5 explains this step in greater detail.

STEP 2: (called ShiftRows for shifting the rows of the state array during the forward process) (The corresponding transformation
Figure 3: One round of encryption is shown at left and one round of decryption at right. (This figure is from Lecture 8 of “Computer and Network Security” by Avi Kak)
during decryption is denoted **InvShiftRows** for Inverse Shift-Row Transformation.)

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

This step is explained in greater detail in Section 8.6.

**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 position** of the ciphertext block of 128 bits. [The phrasing of this last sentence is important. The sentence does **NOT** say that if you change one bit of the plaintext, the algorithm is guaranteed to change every bit of the ciphertext. (Changing every bit of the ciphertext would amount to reversing every bit of the block.) Since a bit can take on only two values, on
the average there will be many bits of the ciphertext that will be identical to the plaintext bits in the same positions after you have changed one bit of the plaintext. However, again on the average, when you change one bit of the plaintext, you will see its effect spanning all of the 128 bits of the ciphertext block. On the other hand, with DES, changing one bit of the plaintext affects only 31 bit positions on the average.]

Section 8.7 explains this step in greater detail.

**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 **InvAddRoundKey** for inverse add round key transformation.)
8.5: THE SUBSTITUTE BYTES STEP: SubBytes and InvSubBytes

• 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 $16 \times 16$.

• 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 $16 \times 16$ lookup table.

• As explained in Section 8.5.1, the entries in the lookup table are constructed by a combination of $GF(2^8)$ arithmetic and bit scrambling.

• The goal of the substitution step is to reduce the correlation between the input bits and the output bits.
at the byte level. The bit scrambling part of the substitution step ensures that the substitution cannot be described in the form of evaluating a simple mathematical function.
8.5.1: Constructing the $16 \times 16$ Lookup Tables for SubBytes and InvSubBytes

- We first fill each cell of the $16 \times 16$ table with the byte obtained by joining together its row index and the column index. [The row index of this table runs from hex 0 through hex F. Likewise, the column index runs from hex 0 through hex F.]

- For example, for the cell located at row index 2 and column indexed 7, we place hex $0x27$ in the cell. So at this point the table will look like

<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>....</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
</tbody>
</table>

- 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 hex value 0x00 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.] [If you are creating your own Python implementation for AES and using the BitVector module, you can use the function \texttt{gf\_MI()} of that module to calculate the multiplicative inverses required for this table.]

- After the above step, 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 scrambling, we next apply the following transformation to each \(b_i\) 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 \(i^{th}\) bit of a specially designated byte \(c\) whose hex
value is \( \text{0x63. } (c_7c_6c_5c_4c_3c_2c_1c_0 \equiv 01100011) \)

- The above bit-scrambling step is 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 \([A]f + \vec{b}\) appearance of this transformation, it is commonly referred to as the affine transformation.]

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

- The very important role played by the \(c\) byte of value \(\text{0x63}\): Consider the following two conditions on the SubBytes step: (1) In order for the byte substitution step to be invertible, the byte-to-byte mapping given to us by the \(16 \times 16\) table must be one-one. That is, for each input byte, there must be a unique output byte. And, to each output byte there must correspond only one input byte. (2) No input byte should map to itself, since a byte mapping to itself would weaken the cipher. Taking multiplicative inverses in the construction of the table does give us unique entries in the table for each input byte — except for
the input byte 0x00 since there is no MI defined for the all-zeros byte. (See Lecture 4 for why that is the case.) If it were not for the c byte, the bit scrambling step would also leave the input byte 0x00 unchanged. With the affine mapping shown above, the 0x00 input byte is mapped to 0x63. At the same time, it preserves the one-one mapping for all other bytes.

- In addition to ensuring that every input byte is mapped to a different and unique output byte, the bit-scrambling step also breaks the correlation between the bits before the substitution and the bits after the substitution.

- 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 steps that go into constructing the 16 × 16 lookup table are reversed for the decryption table, meaning that you first apply the reverse of the bit-scrambling operation to each byte, as explained in the next step, and then you take its multiplicative inverse in GF(2^8).

- For bit scrambling for decryption, 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 $i^{th}$ bit of a specially designated byte $d$ whose hex value is 0x05. ($d_7d_6d_5d_4d_3d_2d_1ddc_0 = 00000101$) Finally, you replace the byte in the cell by its multiplicative inverse in $GF(2^8)$. [IMPORTANT: You might ask whether decryption bit scrambling also maps 0x00 to its constant $d$. No that does not happen. For decryption, the goal of bit scrambling is to reverse the effect of bit scrambling on the encryption side. The bit scrambling operation for decryption maps 0x00 to 0x52.]

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

- If you are still confused about how all the steps come together in the construction of the $16 \times 16$ SBox tables — especially if you are confused about how to write code for the bit scrambling step — the rest of this section shows my Python and Perl implementations of these steps. You will be surprised how easy it is to write this code if you are using the \texttt{BitVector} module in Python and the \texttt{Algorithm::BitVector} module in Perl.

- As you see, the core logic for constructing the S-Boxes can be implemented with just half a dozen lines of code in either Python or Perl.
#!/usr/bin/env python
## gen_tables.py
## Avi Kak (February 15, 2015)
##
## This is the Python implementation of the explanation in Section 8.5.1 of Lecture
## 8. The goal here is to construct two 16x16 lookup tables for byte substitution,
## one for the SubBytes step of the AES algorithm, and the other for the InvSubBytes
## step.

import sys
from BitVector import *

AES_modulus = BitVector(bitstring='100011011')
subBytesTable = []  # SBox for encryption
invSubBytesTable = []  # SBox for decryption

def genTables():
    c = BitVector(bitstring='01100011')
    d = BitVector(bitstring='00000101')
    for i in range(0, 256):
        # For the encryption SBox
        a = BitVector(intVal = i, size=8).gf_MI(AES_modulus, 8) if i != 0 else BitVector(intVal=0)
        # For bit scrambling for the encryption SBox entries:
        a1,a2,a3,a4 = [a.deep_copy() for x in range(4)]
        a ^= (a1 >> 4) ^ (a2 >> 5) ^ (a3 >> 6) ^ (a4 >> 7) ^ c
        subBytesTable.append(int(a))
        # For the decryption Sbox:
        b = BitVector(intVal = i, size=8)
        # For bit scrambling for the decryption SBox entries:
        b1,b2,b3 = [b.deep_copy() for x in range(3)]
        b = (b1 >> 2) ^ (b2 >> 5) ^ (b3 >> 7) ^ d
        check = b.gf_MI(AES_modulus, 8)
        b = check if isinstance(check, BitVector) else 0
        invSubBytesTable.append(int(b))

genTables()
print "SBox for Encryption:"
print subBytesTable
print "\nSBox for Decryption:"
print invSubBytesTable

And shown below is the Perl implementation for doing the same thing:
#!/usr/bin/perl -w

## gen_tables.pl
## Avi Kak (February 16, 2015)
## This is the Perl implementation of the explanation in Section 8.5.1 of Lecture 8.
## The goal here is to construct two 16x16 lookup tables for byte substitution, one
## for the SubBytes step of the AES algorithm, and the other for the InvSubBytes
## step.

use strict;
use Algorithm::BitVector;

my $AES_modulus = Algorithm::BitVector->new(bitstring => '100011011');

my @subBytesTable; # SBox for encryption
my @invSubBytesTable; # SBox for decryption

sub genTables {
    my $c = Algorithm::BitVector->new(bitstring => '01100011');
    my $d = Algorithm::BitVector->new(bitstring => '00000101');
    foreach my $i (0..255) {
        # For the encryption SBox:
        my $a = $i == 0 ? Algorithm::BitVector->new(intVal => 0) :
            Algorithm::BitVector->new(intVal => $i, size => 8)->gf_MI($AES_modulus, 8);
        # For bit scrambling for the encryption SBox entries:
        my ($a1,$a2,$a3,$a4) = map $a->deep_copy(), 0 .. 3;
        $a ^= ($a1 >> 4) ^ ($a2 >> 5) ^ ($a3 >> 6) ^ ($a4 >> 7) ^ $c;
        push @subBytesTable, int($a);
        # For the decryption Sbox:
        my $b = Algorithm::BitVector->new(intVal => $i, size => 8);
        # For bit scrambling for the decryption SBox entries:
        my ($b1,$b2,$b3) = map $b->deep_copy(), 0 .. 2;
        $b = ($b1 >> 2) ^ ($b2 >> 5) ^ ($b3 >> 7) ^ $d;
        my $check = $b->gf_MI($AES_modulus, 8);
        $b = ref($check) eq 'Algorithm::BitVector' ? $check : 0;
        push @invSubBytesTable, int($b);
    }
}

genTables();
print "SBox for Encryption:
";
print "@subBytesTable
";
print "\nSBox for Decryption:
";
print "@invSubBytesTable\n";
The encryption S-Box that a correct implementation should return is shown below: (Note that the values are shown as decimal integers)

\begin{verbatim}
\end{verbatim}

And the decryption S-Box that a correct implementation should return is shown below (again as decimal integers):

\begin{verbatim}
\end{verbatim}

The Python and Perl scripts in this section can be downloaded from the link associated with Lecture 8 at the “Lecture Notes” website.
8.6: THE SHIFT ROWS STEP: ShiftRows and InvShiftRows

• 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.

\[
\begin{bmatrix}
  s_{0.0} & s_{0.1} & s_{0.2} & s_{0.3} \\
  s_{1.0} & s_{1.1} & s_{1.2} & s_{1.3} \\
  s_{2.0} & s_{2.1} & s_{2.2} & s_{2.3} \\
  s_{3.0} & s_{3.1} & s_{3.2} & s_{3.3}
\end{bmatrix}
\quad\rightarrow\quad
\begin{bmatrix}
  s_{0.0} & s_{0.1} & s_{0.2} & s_{0.3} \\
  s_{1.3} & s_{1.0} & s_{1.1} & s_{1.2} \\
  s_{2.2} & s_{2.3} & s_{2.0} & s_{2.1} \\
  s_{3.1} & s_{3.2} & s_{3.3} & s_{3.0}
\end{bmatrix}
\]
8.7: THE MIX COLUMNS STEP: MixColumns and InvMixColumns

- 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 multiplications implied by the word ‘times’ and the additions implied by the word ‘plus’ are meant to be carried out in $GF(2^8)$ arithmetic, as explained in Lecture 7. If you are using the BitVector module in Python, it gives you the method `gf_multiply_modular()` for carrying out such multiplications. The additions are merely XOR operations, as you should know from Lecture 7. The Perl programmers can do the same thing with the Algorithm::BitVector module.] 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. [Note that 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
\[ s'_{0,j} = (0x02 \times s_{0,j}) \otimes (0x03 \times s_{1,j}) \otimes s_{2,j} \otimes s_{3,j} \]

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

\[ s'_{1,j} = s_{0,j} \otimes (0x02 \times s_{1,j}) \otimes (0x03 \times s_{2,j}) \otimes s_{3,j} \]

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

\[ s'_{2,j} = s_{0,j} \otimes s_{1,j} \otimes (0x02 \times s_{2,j}) \otimes (0x03 \times s_{3,j}) \]

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

\[ s'_{3,j} = (0x03 \times s_{0,j}) \otimes s_{1,j} \otimes s_{2,j} \otimes (0x02 \times s_{3,j}) \]

- 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

• Each round has its own round key that is derived from the original 128-bit encryption key in the manner described in this section. One of the four steps of each round, for both encryption and decryption, involves XORing of the round key with the state array.

• The AES Key Expansion algorithm is used to derive the 128-bit round key for each round from the original 128-bit encryption key. As you’ll see, the logic of the key expansion algorithm is designed to ensure that if you change one bit of the encryption key, it should affect the round keys for several rounds.

• 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 $4 \times 4$ array of bytes, as shown at the top of the next page.
\[ \begin{bmatrix}
  k_0 & k_4 & k_8 & k_{12} \\
  k_1 & k_5 & k_9 & k_{13} \\
  k_2 & k_6 & k_{10} & k_{14} \\
  k_3 & k_7 & k_{11} & k_{15}
\end{bmatrix} \]

\[ \downarrow \]

\[ \begin{bmatrix}
  w_0 & w_1 & w_2 & w_3
\end{bmatrix} \]

- 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 \( w_0, w_1, w_2, w_3, \ldots, w_{43} \).

- 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 \( w_0, w_1, w_2, w_3, w_4, w_5, \ldots, w_{43} \)?

• The key expansion algorithm will be explained in the next subsection with the help of Figure 4. As shown in the figure, 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: *The key expansion takes place on a four-word to four-word basis as shown here.* (This figure is from Lecture 8 of “Computer and Network Security” by Avi Kak)
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 $i^{th}$ round:

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

For these to serve as the round key for the $i^{th}$ 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

\[
\begin{align*}
  w_{i+5} & = w_{i+4} \otimes w_{i+1} \\
  w_{i+6} & = w_{i+5} \otimes w_{i+2} \\
  w_{i+7} & = w_{i+6} \otimes w_{i+3}
\end{align*}
\]

(1) (2) (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:

\[
w_{i+4} = w_i \otimes g(w_{i+3})
\]

(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 $16 \times 16$ lookup table as used in the SubBytes step of the encryption rounds. [The SubBytes step was explained in Section 8.5]

- 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 $i^{th}$ 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 $i^{th}$ round. The right hand side of the equation says that the rightmost three bytes of the round constant are zero.

$$Rcon[i] = (RC[i], \text{0x}00, \text{0x}00, \text{0x}00)$$

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

$$RC[1] = \text{0x}01$$
$$RC[j] = \text{0x}02 \times RC[j-1]$$
Recall from Lecture 7 that multiplication by $0x02$ amounts to multiplying the polynomial corresponding to the bit pattern $RC[j-1]$ by $x$.

- 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.

- The presentation of the key expansion algorithm so far in this section was 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 either of the two other possibilities for key lengths: 192 bits and 256 bits. A key length of 192 bits entails 12 rounds and a key length of 256 bits entails 14 rounds. (However, 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.

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 $j^{th}$ 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 the 128-bit key 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.

- 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: DIFFERENTIAL, LINEAR, AND INTERPOLATION ATTACKS ON BLOCK CIPHERS

• This section is for a reader who is curious as to why the substitution step in AES involves taking the MI of each byte in $GF(2^8)$ and bit scrambling. As you might have realized already, that is the only nonlinear step in mapping a plaintext block to a ciphertext block in AES.

• Back in the 1990’s (this is the decade preceding the development of the Rijndael cipher which is the precursor to the AES standard) there was much interest in investigating the block ciphers of the day (DES being the most prominent) from the standpoint of their vulnerabilities to differential and linear cryptanalysis. **The MI byte substitution step in AES is meant to protect it against such cryptanalysis.** At around the same time, it was shown by Jakobsen and Knudsen in 1997 that block ciphers whose SBoxes were based on polynomial arithmetic in Galois fields could be vulnerable to a new attack that they referred to as the **interpolation attack.** The bit scrambling part of the SBox in AES is meant to be a protection against the interpolation attack. [As mentioned earlier in Section 3.2.2 of Lecture 3, the differential attack was first described]
by Biham and Shamir in a paper titled “Differential Cryptanalysis of DES-like Cryptosystems” that appeared in the Journal of Cryptology in 1991. The linear attack was first described by Matsui in a publication titled “Linear Cryptanalysis Method for DES Ciphers,” in “Lecture Notes in Computer Science, no. 764. Finally, the interpolation attack was first described by Jakobsen and Knudsen in a publication titled “The Interpolation Attack on Block Ciphers” that appeared in Lecture Notes in Computer Science, Haifa, 1997.

Therefore, in order to fully appreciate the SBox in AES, you have to have some understanding of these three forms of cryptanalysis. The phrases “differential cryptanalysis” and “linear cryptanalysis” are synonymous with “differential attack” and “linear attack.”

The rest of this section reviews these three attacks briefly. [You will get more out of this section if you first read the tutorial ”A Tutorial on Linear and Cryptanalysis” by Howard Heys of the Memorial University of Newfoundland. Googling that author’s name will take you directly to the tutorial.]

Starting our discussion with the **differential attack**, it is based on the following concepts:

- How a differential (meaning an XOR of two bit blocks) propagates through a sequence of rounds is independent of the round keys. [As you’ll recall from the note in small-font blue in Section 3.3.2 of Lecture 3, differential cryptanalysis is a *chosen plaintext attack* in which the attacker feeds plaintext bit blocks pairs, $X_1$ and $X_2$, with known differences $\Delta X = X_1 \oplus X_2$ be-
between them, into the cipher while observing the differences $\Delta Y = Y_1 \otimes Y_2$ between the corresponding ciphertext blocks. We refer to $\Delta X$ as the input differential and $\Delta Y$ as the output differential. The fact that the propagation of a differential is NOT affected by the round keys can be established in the following manner: Consider just one round and let’s say that $K$ is the round key. Let’s further say that the output of the round is what is produced by the SBox XOR’ed with the round key. For two different inputs $X_1$ and $X_2$ to the round, let $Y_1'$ and $Y_2'$ denote the outputs of the SBox and and let $Y_1$ and $Y_2$ denote the final output of the round. We have $Y_1 = K \otimes Y_1'$ and $Y_2 = K \otimes Y_2'$. The differential $\Delta Y = Y_1 \otimes Y_2$ for the output after key mixing is related to the other differentials by $\Delta Y = Y_1 \otimes Y_2 = K \otimes Y_1' \otimes K \otimes Y_2' = Y_1' \otimes Y_2'$. Therefore, the mapping between the input and the output differentials of a round is not a function of the round key.

– If one is not careful, the byte substitution step in an SBox can create significant correlations between the input differentials and the output differentials.

– The correlations between the input differentials and the output differentials, when they are significant, can be exploited to make good guesses for the bits of the last round key.

• Therefore, our first order of business is to understand the relationship between the input and the output differentials for a given choice of the SBox.

• The Perl script shown next, find_differentials_correlations.pl, cal-
calculates a 2D histogram of the relationship between the input and the output differentials. The statements in lines (B8) and (B9), with one of the lines commented-out, give you two choices for the operation of the SBox. If you use the statement in line (B8), the byte substitutions will consist of replacing each byte by its MI in $GF(2^8)$ that is based on the AES modulus. On the other hand, if you use the currently commented-out statement in line (B9), the byte substitution will take place according to the lookup table supplied through line (A9). [Yes, to be precise, the MI based byte substitution could also be carried out through a lookup table. That is, just because one SBox is based on MI calculations and the other on looking up a table is NOT the fundamental difference between the two. The lookup table supplied through line (A9) was arrived at by experimenting with several such choices made possible by the commented out statements in lines (A7) and (A8). The call to `shuffle()` in line (A7) gives a pseudorandom permutation of the 256 one-byte words. Based on a dozen runs of the script, the permutation shown in line (A9) yielded the best inhomogeneous histogram for the input/output differentials. The reader may wish to carry out such experiments on his/her own and possibly make a different choice for the lookup table in line (A9).]

- The portion of the script starting with line (F1) is just for displaying the histogram of the input/output differentials and, therefore, not central to understand what we mean by the differentials here and the correlations between the input differentials and the output differentials.

```bash
#!/usr/bin/perl -w
## find_differentials_correlations.pl
## Avi Kak (March 4, 2015)
```

45
## This script creates a histogram of the mapping between the input differentials and the output differentials for an SBox. You have two choices for the SBox --- as reflected by lines (B8) and (B9) of the script. For a given input byte, the statement in line (B8) returns the MI (multiplicative inverse) of the byte in GF(2^8) based on the AES modulus. And the statement in line (B8) returns a byte through a user-specified table lookup. The table for this is specified in line (A9). More generally, such a table can be created by a random permutation through the commented-out statements in lines (A7) and (A8).

```perl
use strict;
use Algorithm::BitVector;
use Graphics::GnuplotIF;
$|++; 

my $debug = 1;
my $AES_modulus = Algorithm::BitVector->new(bitstring => '100011011'); #(A1)

my $M = 64; # CHANGE THIS TO 256 FOR A COMPLETE CALCULATION #(A2)
# This parameter control the range of inputs
# bytes for creating the differentials. With
# its value set to 64, only the differentials
# for the bytes whose int values are between 0
# and 63 are tried.

# Initialize the histogram:
my $differential_hist; #(A3)
foreach my $i (0..255) { #(A4)
    foreach my $j (0..255) { #(A5)
        $differential_hist->{[$i][$j]} = 0; #(A6)
    }
}

# When SBox is based on lookup, we will use the "table" created by randomly
# permuting the the number from 0 to 255:
#my $lookuptable = shuffle([0..255]); #(A7)
#my $lookuptable = @lookuptable; #(A8)
my @lookuptable = qw(213 170 104 116 66 14 16 76 219 200 42 22 17 241 197 41 216 85 140 183 244 235 6 118 208 74 218 99 44 1 89 11 205 195 125 47 236 113 237 131 109 102 9 21 220 59 154 119 148 38 120 13 217 16 100 191 81 240 196 122 83 177 229 142 35 88 48 167 0 29 153 163 146 166 77 99 43 10 194 232 189 238 164 204 111 69 51 126 62 211 242 70 214 247 55 202 78 239 114 184 112 228 84 152 187 45 49 175 58 253 72 95 19 37 73 145 87 198 71 159 34 91 168 250 255 8 121 96 50 141 181 67 26 243 130 68 61 24 105 210 172 139 136 157 133 80 93 39 2 143 161 186 33 144 178 30 92 138 169 86 249 252 155 193 63 223 203 245 129 4 171 115 3 40 151 7 188 231 174 25 23 207 180 56 46 206 215 227 162 199 97 147 182 149 108 36 132 5 12 103 110 209 160 137 53 224 185 173 20 222 246 28 179 134 75 254 57 60 234 52 165 225 248 31 230 156 124 233 158 27 18 94 65 32 54 106 192 221 190 101 98 251 212 150 201 117 127 107 176 226 135 123 82 15 64 90); #(A9)

# This call creates the 2D plaintext/ciphertext differential histogram:
gen_differential_histogram(); #(A10)

# The call shown below will show that part of the histogram for which both
```
# the input and the output differentials are in the range (32, 63).
display_portion_of_histogram(32, 64);  #(A11)

plot_portion_of_histogram($differential_hist, 32, 64);  # (A12)
## The following call makes a hardcopy of the plot:
plot_portion_of_histogram($differential_hist, 32, 64, 3);  # (A13)

sub gen_differential_histogram {  # (B1)
  foreach my $i (0 .. $M-1) {  # (B2)
    print "\n\ni=$i\n" if $debug;  # (B3)
    foreach my $j (0 .. $M-1) {  # (B4)
      print ". " if $debug;  # (B5)
      my ($a, $b) = (Algorithm::BitVector->new(intVal => $i, size => 8),
                      Algorithm::BitVector->new(intVal => $j, size => 8));  # (B6)
      my $input_differential = int($a ^ $b);  # (B7)
      # Of the two statements shown below, you must comment out one depending
      # on what type of an SBox you want:
      my ($c, $d) = (get_sbox_output_MI($a), get_sbox_output_MI($b));  # (B8)
      # my ($c, $d) = (get_sbox_output_lookup($a), get_sbox_output_lookup($b));  # (B9)
      my $output_differential = int($c ^ $d);  # (B10)
      $differential_hist->[\$input_differential][\$output_differential]++;  # (B11)
    }
  }
}

sub get_sbox_output_MI {  # (C1)
  my $in = shift;  # (C2)
  return int($in) != 0 ? $in->gf_MI($AES_modulus, 8) :  # (C3)
    Algorithm::BitVector->new(intVal => 0);  # (C4)
}

sub get_sbox_output_lookup {  # (D1)
  my $in = shift;  # (D2)
  return Algorithm::BitVector->new(intVal => $lookuptable[int($in)], size => 8);  # (D3)
}

# Fisher-Yates shuffle:
sub shuffle {  # (E1)
  my $arr_ref = shift;  # (E2)
  my $i = @$arr_ref;  # (E3)
  while ( $i-- ) {  # (E4)
    my $j = int rand( $i + 1 );  # (E5)
    @$arr_ref[ $i, $j ] = @$arr_ref[ $j, $i ];  # (E6)
  }
  return $arr_ref;  # (E7)
}

############################################################################### Support Routines for Displaying the Histogram ################################################################################

# Displays in your terminal window the bin counts in the two-dimensional histogram
# for the input/output mapping of the differentials. You can control the portion of
# the 2D histogram that is output by using the first argument to set the lower bin
# index and the second argument the upper bin index along both dimensions.
# Therefore, what you see is always a square portion of the overall histogram.
sub display_portion_of_histogram {  # (F1)
my $lower = shift;  # (F2)
my $upper = shift;  # (F3)
foreach my $i ($lower .. $upper - 1) {
    print "\n";  # (F4)
    foreach my $j ($lower .. $upper - 1) {  # (F5)
        print "differential_hist->[\$i][\$j] \n";  # (F6)
    }
}

# Displays with a 3-dimensional plot a square portion of the histogram. Along both
# the X and the Y directions, the lower bound on the bin index is supplied by the
# SECOND argument and the upper bound by the THIRD argument. The last argument is
# needed only if you want to make a hardcopy of the plot. The last argument is set
# to the number of second the plot will be flashed in the terminal screen before it
# is dumped into a '.png' file.
sub plot_portion_of_histogram {
    my $hist = shift;  # (G1)
    my $lower = shift;  # (G2)
    my $upper = shift;  # (G3)
    my $pause_time = shift;  # (G4)
    my @plot_points = ();  # (G5)
    my $bin_width = my $bin_height = 1.0;  # (G6)
    my ($x_min, $y_min, $x_max, $y_max) = ($lower, $lower, $upper, $upper);  # (G7)
    foreach my $y ($y_min..$y_max-1) {  # (G8)
        foreach my $x ($x_min..$x_max-1) {  # (G9)
            push @plot_points, [$x, $y, $hist->[$y][$x]];  # (G10)
        }
    }
    @plot_points = sort {$a->[0] <=> $b->[0]} @plot_points;  # (G11)
    @plot_points = sort {$a->[1] <=> $b->[1] if $a->[0] == $b->[0]} @plot_points;  # (G12)
    my $temp_file = "__temp.dat";  # (G13)
    open(OUTFILE, ">$temp_file") or die "Cannot open temporary file: $!";  # (G14)
    my ($first, $oldfirst);  # (G15)
    $oldfirst = $plot_points[0]->[0];  # (G16)
    foreach my $sample (@plot_points) {  # (G17)
        $first = $sample->[0];  # (G18)
        if ($first == $oldfirst) {  # (G19)
            my @out_sample;  # (G20)
            $out_sample[0] = $sample->[0];  # (G21)
            $out_sample[1] = $sample->[1];  # (G22)
            $out_sample[2] = $sample->[2];  # (G23)
            print OUTFILE "@out_sample\n";  # (G24)
        } else {  # (G25)
            print OUTFILE "\n";  # (G26)
        }
        $oldfirst = $first;  # (G27)
    }
    print OUTFILE "\n";  # (G28)
    close OUTFILE;
    my $argstring = "END";  # (G28)
    set xrange [$x_min:$x_max]
    set yrange [$y_min:$y_max]
    set view 80,15
    set hidden3d
For an accurate and complete calculation of the input/output differentials histogram, you’d need to change the value of $M$ in line (A2) to 256. That would result in a large $256 \times 256$ histogram of integer values. For the purpose of our explanation here, we will make do with $M = 64$. The resulting histogram would not be an accurate depiction of the reality. Nonetheless, it will suffice for the purpose of the explanation that follows.

If you run the script with the SBox as specified in line (B8), you will end up with a display of numbers as shown below for the portion of the differentials histogram that is bounded by bin index values ranging from 32 to 63 in both directions. To understand these values, let’s look at the first nonzero entry in the first row, which happens to be in the column indexed 40. Recognizing that the first row corresponds to the bin index 32, that nonzero count of 2 means that in all of the runs of the loop in lines (B1) through (B11) of the script, there were 2 cases when the input
differential was $\Delta X = 00100000$ (integer value = 32) and the output differential was $\Delta Y = 00101000$ (integer value = 40).

Note that some of the bin counts in the portion of the histogram shown above are as large as 4. A 3D plot of this portion of the
Figure 5: Shown is a portion of the histogram of the input/output differentials for an SBox consisting of MI in $GF(2^8)$. (This figure is from Lecture 8 of “Computer and Network Security” by Avi Kak)

histogram as shown in Figure 8.5 is meant to make it easier to see how many such bins exist in the portion of the histogram plotted.

- If you comment out line (B8) and uncomment line (B9) to change to the second option for the SBox, the same histogram will look like what is shown in Figure 8.6. Recall that the second option consists of doing byte substitution through the lookup table in line (A9).

- As you can see in Figure 8.6, the second option for the SBox generates a more non-uniform histogram for the input/output
differentials. Ideally, for any input differential $\Delta X$, you would want the output differential $\Delta Y$ to be distributed uniformly with a probability of $1/2^n$ for an $n$-bit block cipher. (This would translate into a count of 1 in every bin except for the $(0, 0)$ bin for reasons explained in the small-font note at the end of this bullet.) That way, the output differentials will give an attacker no clues regarding either the plaintext or the encryption key. However, attaining this ideal is theoretically impossible. 

As to why the theoretical ideal is impossible to attain, let’s first review some of the more noteworthy features of such a histogram: (1) If we had shown the entire histogram, especially if the cell at $(0, 0)$ was included in the histogram display, you would see the largest peak located over the $(0, 0)$ bin and the bin count associated with this peak would be 256. This is a result of the fact that, in the double loop in lines (B1) through (B11) of the script `find_differentials_correlations.pl`, as we scan through 256 different values for the first input byte,
and, for each input byte, as we scan through 256 values for the second input byte, there will be 256 cases when 
the first and the second bytes are identical. Therefore, for these 256 cases, we will have $\Delta X = 0$, and also $\Delta Y = 0$. This would give us a count of 256 over the (0, 0) bin. (2) Another feature that all such histograms 
possess is that every non-zero bin count is even. This is on account of the fact that in the double loop in lines 
(B1) through (B11) of the script, the same $\Delta X$ occurs in multiples of 2 since $\Delta X = X_i \oplus X_j = X_j \oplus X_i$. (3) 
The sum of all the counts in each row and each column must add up to 256. That is because, every differential 
in the input must map one of 256 possible differentials at the output. (4) Therefore, for best such histograms 
(that is, histograms with no biases in how the input and the output differentials are related), half the randomly 
located bins in each row would contain a count of 2 (this would not apply to the bins in the topmost row or 
the leftmost column). (5) For all the reasons stated here, the ideal of having a count of 1 in each bin of the 
$256 \times 256$ bins of the histogram is clearly not achievable — even theoretically.

- As the value of the variable $M$ in line (A2) of the script `find_differentials_correlations.pl` approaches 256, with the MI option for the 
SBox in line (B8), you will see more and more bins showing the 
best possible count of 2 in the histogram of Figure 8.5. On the 
other hand, with the table lookup option in line (B9) for the 
SBox, you will see the histogram in Figure 8.6 staying just as 
non-uniform as it is now — with the max peaks becoming some-
what larger.

- The Perl script that follows, `differential_attack_toy_example.pl`, is a 
demonstration of how the sort of non-uniformities in the his-
togram of the input/output differentials can be exploited to re-
cover some portions of the key for at least the last round of a block 
cipher. However, note that this script is only a toy example just
to get across the ideas involved in mounting a differential attack on a block cipher. The logic presented in this script would not work by any stretch of imagination on any realistic block cipher.

- The script `differential_attack_toy_example.pl` mounts a differential attack on the encryption scheme in lines (C1) through (C14) of the code. The SBox byte substitution is based on table lookup using the table supplied through line (A9). The byte returned by table lookup is XOR’ed with the round key. The round key is shifted circularly by one bit position to the right for the next round key. [For a more realistic simple example of a differential attack that involves both an SBox and permutations in each round, the reader should look up the previously mentioned tutorial “A Tutorial on Linear and Differential Cryptanalysis” by Howard Heys. The block size in that tutorial is 4 bits.]

- The beginning part of the `differential_attack_toy_example.pl` script that follows is the same as in the script `find_differentials_correlations.pl` that you saw earlier in this section. That’s because, as mentioned earlier, a differential attack exploits the predictability of the ciphertext differentials vis-a-vis the plaintext differentials. Therefore, lines (A14) through (A27) of the script are devoted to the calculation of a 2D histogram that measures the joint probabilities of occurrence of the input and the output differentials. As the comment lines explain, note how the information generated is saved on the disk in the form of DBM files. So, as you are experimenting with the attack logic in lines (B1) through (B26)
of the script and running the script over and over, you would not need to generate the plaintext/ciphertext differentials histogram each time. You can start from ground zero (that is, you can re-generate the histogram) at any time provided you first call

```
clean_db_files.pl
```

to clear out the DBM files in your directory. The script `clean_db_files.pl` is included in the code you can download from the lectures notes website.

- The plaintext/ciphertext differentials histogram is converted into the hash `%worst_differentials` in line (A22). In case you are wondering why we couldn’t make do with the disk-based `%worst_differentials_db` hash that is defined in line (A14), it is because the latter does not support the `exists()` function that we need in line (B10) of the script. The keys in both these hashes are the plaintext differentials and, for each key, the value the ciphertext differential where the histogram count exceeds the specified threshold. [Potential source of confusion: Please do not confuse the use of “key” as in the `<key,value>` pairs that are stored in a Perl hash with the use of key as in “encryption key.”]

- Finally, we mount the attack in line (A29). The attack itself is implemented in lines (B1) through (B26). If you only specify one round in line (A2), the goal of the attack would be estimate the encryption key as specified by line (A3). However, if the number
of rounds exceeds 1, the goal of the attack is to estimate the key in the last round key. The attack logic consists simply of scanning through all possible plaintext differentials and using only those that form the keys in the \%worst_differentials hash, finding the corresponding the ciphertext differentials. Once we have chosen a plaintext pair, and, therefore a plaintext differential, in line (B8), we apply partial decryption to the corresponding ciphertext bytes in lines (B21) and (B22). Subsequently, in line (B24), we check whether the differential formed by the two partial decryptions exists in our \%worst_differentials hash for each candidate last-round key. If this condition is satisfied, a vote is cast for that candidate key.

```perl
#!/usr/bin/perl -w

## differential_attack_toy_example.pl
## Avi Kak (March 4, 2015)

## This script is a toy example to illustrate some of the key elements of a
differential attack on a block cipher.

## We assume that our block size is one byte and the SBox consists of finding a
## substitute byte by table lookup. We further assume that each round consists of
## one byte substitution step followed by xor'ing the substituted byte with the
## round key. The round key is the encryption key that is circularly shifted to the
## right by one position for each round.

## Since you are likely to run this script repeatedly as you experiment with
## different strategies for estimating the subkey used in the last round, the script
## makes it easy to do so by writing the information that is likely to stay constant
## from one run to the next to disk-based DBM files. The script creates the
## following DBM files:
##
## worst_differentials.dir and worst_differentials.pag -- See Line (A14)
##
## These DBM files are created the very first time you run this script. Your
## subsequent runs of this script will be much faster since this DBM database
## would not need to be created again. Should there be a need to run the script
## starting from ground zero, you can clear the DBM files created in your directory
```
## by calling the script:
##   clean_db_files.pl
## Finally, if you set the number of tries in Line (A10) to a large number and you
## are tired of waiting, you can kill the script at any time you wish. To see the
## vote counts accumulated up to that point for the different possible candidates
## for the last round key, just run the script:
##   get_vote_counts.pl
## The scripts clean_db_files.pl and get_vote_counts.pl are in the gzipped archive
## that goes with Lecture 8 at the lecture notes web site.

use strict;
use Algorithm::BitVector;
$|++; 

my $debug = 1;
my $AES_modulus = Algorithm::BitVector->new(bitstring => '100011011');  # (A1)
my $number_of_rounds = 1;  # (A2)
my $encryption_key = Algorithm::BitVector->new(bitstring => '10001011');  # (A3)
my $differential_hist;  # (A4)
my %decryption_inverses;  # (A5)
my %worst_differentials;  # (A6)
my @worst_input_differentials;  # (A7)
my @worst_output_differentials;  # (A8)

my $hist_threshold = 8;  # (A9)
my $tries = 500;  # (A10)

unlink glob "votes.*";  # (A11)
dbmopen my %votes_for_keys, "votes", 0644
  or die "cannot create DBM file: $!";  # (A12)

# This lookup table is used for the byte substitution step during encryption in the
# subroutine defined in lines (C1) through (C14). By experimenting with the script
# differentials_frequency_calculator.pl this lookup table was found to yield a good
# non-uniform histogram for the plaintext/ciphertext differentials.
my @lookup_table = qw(213 170 104 116 66 14 76 219 200 42 22 17 241 197 41 216 85 140
  183 244 235 6 118 208 74 218 99 44 1 89 11 205 195 125 47 236 113
  237 131 109 102 9 21 220 59 154 119 148 38 120 13 217 16 100 191 81
  240 196 122 83 177 229 142 35 88 48 167 0 29 153 163 146 166 77 79
  43 10 194 232 189 238 164 204 111 69 51 126 62 211 242 70 214 247 55
  202 78 239 114 184 112 228 84 152 187 45 49 175 58 253 72 95 19 37
  73 145 87 198 71 159 34 91 168 250 255 8 121 96 50 141 181 67 26 243
  130 68 61 24 105 210 172 139 136 128 157 133 80 93 39 2 143 161 186 33
  144 178 30 92 138 169 86 249 252 155 193 63 223 203 245 129 4 171
  115 3 40 151 7 188 231 174 25 23 207 180 56 46 206 215 227 162 199
  97 147 182 149 108 36 132 5 12 103 110 209 160 137 53 224 185 173
  20 222 246 28 179 134 75 254 57 60 234 52 165 225 248 31 230 156
  124 233 158 27 1 18 94 65 32 54 106 192 221 190 101 98 251 212 150
  201 117 127 107 176 226 135 123 82 15 64 90);  # (A13)

# In what follows, we first check if the worst_differentials DBM files were created
# previously by this script. If they are already on the disk, create the disk-based
# hash %worst_differentials_db from the data in those files. If not, create the DBM
# files so that they can subsequently be populated by the call in line (A18).
# [IMPORTANT: In a more realistic attack logic, you will need to create a more
# general version of the code in lines (A14) through (A21) so that you find the
# histogram for the plaintext/ciphertext differentials not for just one round, but
# for all the rounds involved. See the tutorial by Howard Heys for this important
# point.]

dbmopen my %worst_differentials_db, "worst_differentials", 0644
or die "Can't open DBM file: $!"; #(A14)
unless (keys %worst_differentials_db) { #(A15)
    foreach my $i (0..255) { #(A16)
        foreach my $j (0..255) { #(A17)
            $differential_hist->[[$i][$j] = 0;
            #(A18)
        }
    }
    gen_differential_histogram(); #(A19)
    # The call shown below will show that part of the histogram for which both
    # the input and the output differentials are in the range (32, 63).
    display_portion_of_histogram(32, 64) if $debug; #(A20)
    # From the 2D input/output histogram for the differentials, now represent that
    # information has a hash in which the keys are the plaintext differentials and
    # the value associated with each key the ciphertext differential whose histogram
    # count exceeds the supplied threshold:
    find_most_probable_differentials($hist_threshold);#(A21)
    %worst_differentials = %worst_differentials_db; #(A22)
    die "no candidates for differentials: $!" if keys %worst_differentials == 0; #(A23)
    @worst_input_differentials = sort {$a <=> $b} keys %worst_differentials; #(A24)
    @worst_output_differentials = %worst_differentials{@worst_input_differentials}; #(A25)
    if ($debug) {
        print "worst input differentials: @worst_input_differentials\n";
        print "worst output differentials: @worst_output_differentials\n";
    }#(A26)
}

%worst_differentials = %worst_differentials_db; #(A22)
die "no candidates for differentials: $!" if keys %worst_differentials == 0; #(A23)
@worst_input_differentials = sort {$a <=> $b} keys %worst_differentials; #(A24)
@worst_output_differentials = %worst_differentials{@worst_input_differentials}; #(A25)
if ($debug) {
    print "\nworst input differentials: @worst_input_differentials\n";
    print "\nworst output differentials: @worst_output_differentials\n";
}#(A26)

# The following call makes a hash that does the opposite of what is achieved by
# indexing into the lookup table of line (A13). It fills the hash
# 'decryption_inverses' with <key,value> pairs, with the keys being the ciphertext
# bytes and the values being the corresponding plaintext bytes.
find_inverses_for_decryption(); #(A28)

estimate_last_round_key(); #(A29)

# Now print out the ten most voted for keys. To see the votes for all possible keys,
# execute the script get_vote_counts.pl separately after running this script.
print "no votes for any candidates for the last round key\n" if keys %votes_for_keys == 0; #(A30)
if (scalar keys %votes_for_keys) {
    my @vote_sorted_keys =
        sort {$votes_for_keys{$b} <=> $votes_for_keys{$a}} keys %votes_for_keys; #(A32)
    print "Displaying the keys with the largest number of votes: @vote_sorted_keys[0..9]\n";
}#(A33)

# Subroutines

# The differential attack:
sub estimate_last_round_key {
  my $attempted = 0;
  foreach my $i (2..255) {
    print "+ " if $debug;
    my $plaintext1 = Algorithm::BitVector->new(intVal => $i, size => 8);
    foreach my $j (2..255) {
      my $plaintext2 = Algorithm::BitVector->new(intVal => $j, size => 8);
      my $input_differential = $plaintext1 ^ $plaintext2;
      next if int($input_differential) < 2;
      next unless exists $worst_differentials{int($input_differential)};
      print "- " if $debug;
      my ($ciphertext1, $ciphertext2) = (encrypt($plaintext1, $encryption_key), encrypt($plaintext2, $encryption_key));
      my $output_differential = $ciphertext1 ^ $ciphertext2;
      next if int($output_differential) < 2;
      last if $attempted++ > $tries;
    }
  }
}

sub encrypt {
  my $plaintext = shift; # must be a bitvector
  my $key = shift; # must be a bitvector
  my $round_input = $plaintext;
  my $round_output;
  my $round_key = $key;
  if ($number_of_rounds > 1) {
    foreach my $round (0..$number_of_rounds-1) {
      $round_output = get_sbox_output_lookup($round_input) ^ $round_key;
      $round_input = $round_output;
      $round_key = $round_key >> 1;
    }
  } else {
    $round_output = get_sbox_output_lookup($round_input) ^ $key;
  }
  return $round_output;
}
Since the SubBytes step in encryption involves taking the square of a byte in GF(2^8) based on AES modulus, for invSubBytes step for decryption will involve taking square-roots of the bytes in GF(2^8). This subroutine calculates these square-roots.

```perl
sub find_inverses_for_decryption { # (D1)
    foreach my $i (0 .. @lookuptable - 1) { # (D2)
        $decryption_inverses{$lookuptable[$i]} = $i; # (D3)
    }
}
```

This function represents the histogram of the plaintext/ciphertext differentials in the form of a hash in which the keys are the plaintext differentials and the value for each plaintext differential the ciphertext differential where the histogram count exceeds the threshold.

```perl
sub find_most_probable_differentials { # (F1)
    my $threshold = shift; # (F2)
    foreach my $i (0..255) { # (F3)
        foreach my $j (0..255) { # (F4)
            $worst_differentials_db{$i} = $j if $differential_hist->[[$i]][$j] > $threshold; # (F5)
        }
    }
}
```

This subroutine generates a 2D histogram in which one axis stands for the plaintext differentials and the other axis the ciphertext differentials. The count in each bin is the number of times that particular relationship is seen between the plaintext differentials and the ciphertext differentials.

```perl
sub gen_differential_histogram { # (G1)
    foreach my $i (0 .. 255) { # (G2)
        print "\ngen_differential_hist: i=$i\n" if $debug; # (G3)
        foreach my $j (0 .. 255) { # (G4)
            print ". " if $debug; # (G5)
            my ($a, $b) = (Algorithm::BitVector->new(intVal => $i, size => 8), # (G6)
                          Algorithm::BitVector->new(intVal => $j, size => 8)); # (G7)
            my $input_differential = int($a ^ $b); # (G8)
            my ($c, $d) = (get_sbox_output_lookup($a), get_sbox_output_lookup($b)); # (G9)
            my $output_differential = int($c ^ $d); # (G10)
            $differential_hist->[[$input_differential]][$output_differential]++; # (G11)
        }
    }
}
```

```perl
sub get_sbox_output_lookup { # (D1)
    my $in = shift; # (D2)
    return Algorithm::BitVector->new(intVal => $lookuptable[int($in)], size => 8); # (D3)
}
```

Displays in your terminal window the bin counts in the two-dimensional histogram for the input/output mapping of the differentials. You can control the portion of the 2D histogram that is output by using the first argument to set the lower bin index and the second argument the upper bin index along both dimensions. Therefore, what you see is always a square portion of the overall histogram.

```perl
sub display_portion_of_histogram { # (J1)
    my $lower = shift; # (J2)
    my $upper = shift; # (J3)
    # Displays in your terminal window the bin counts in the two-dimensional histogram for the input/output mapping of the differentials. You can control the portion of the 2D histogram that is output by using the first argument to set the lower bin index and the second argument the upper bin index along both dimensions. Therefore, what you see is always a square portion of the overall histogram.
}
When you run the script for just one round, that is, when you set the value of the variable $number_of_rounds to 1 in line (A2), you should get the following answer for the ten encryption keys that received the largest number of notes (in decreasing order of the number of votes received):

139  51  200  225  108  216  161  208  26  140

This answer is how you'd expect it to be since the decimal 139 is equivalent to the binary 10001011, which is the encryption key set in line (A3) of the script. For a more detail look at the distribution of the votes for the keys, execute the script:

get_vote_counts.pl

This script will return an answer like

139: 501  51: 40  200: 40  225: 40  108: 39  ...

where the number following the colon is the number for votes for the integer value of the encryption key shown at the left of the colon.
• If you run the attack script with the number of rounds set to 2 in line (A2), you should see the following answer for the ten keys that received the largest number of votes:

\[ 82, 180, 214, 20, 72, 44, 109, 105, 52, 174 \]

This answer says that the most likely key used in the second round is the integer 82, which translates into the binary 01010010. If you examine the logic of encryption in lines (C1) through (C14) — especially if you focus on how the round key is set in line (C11) — the answer returned by the script is incorrect. However, that is not surprising since our input/output histogram of the differentials is based on just one round. As explained in the previously mentioned tutorial by Howard Heys, we would need to construct a more elaborate model of the differentials propagate through multiple rounds for the script to do better in those cases.

• That brings us to the subject of **linear attacks** on block ciphers. A linear attack on a block cipher is a *known plaintext attack*. In such attacks, the adversary has access to a set of plaintexts and the corresponding ciphertexts. However, unlike the differential attack, the adversary does not choose any specific subset of these.

A linear attack exploits linear relationships between the bits to the input to the SBox and the bits at the output. Let \((X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7)\) represent the bits at the input to an SBox and let \((Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, Y_7)\) represent the bits at the output. Should it be the case that there exist bit positions \(i_1, \ldots, i_m\) at the input and
the bit positions $j_1, \ldots, j_n$ at the output for some values of $m$ and $n$ so that the following is either often true or often false:

\[
\text{The phrase 'often true' here means 'significantly above average' and the phrase 'often false' means 'significantly below average'. For an ideal SBox, such relationships would be true (and, therefore, false) with a probability of 0.5 for all sets of bit positions at the input and the output.}
\]

\[
X_{i_{11}} \otimes X_{i_{12}} \otimes \cdots \otimes X_{i_{1m}} \otimes Y_{i_{11}} \otimes Y_{i_{12}} \otimes \cdots \otimes Y_{i_{1n}} = 0
\]

that fact can be exploited by a linear attack to make good estimates for the key bits in at least the last round of a block cipher.

- As should be obvious to the reader, the linear relationship shown above can also be expressed as

\[
X_{i_{11}} \otimes X_{i_{12}} \otimes \cdots \otimes X_{i_{1m}} = Y_{i_{11}} \otimes Y_{i_{12}} \otimes \cdots \otimes Y_{i_{1n}}
\]

- It is important to realize that such a relationship must hold for all possible values of 0’s and 1’s at the bit positions in question. Consider the case when the list of output bit positions is empty. Now there will be 128 out of 256 different possible bit patterns at the input for which the linear relationship, as shown in Equation (5), will be satisfied. For this case, whenever an input bit pattern has an even number of 1’s, its XOR-sum will be zero. (And that will happen in 128 out of 256 cases.) The same would be the case when we consider an empty set of input bits and all possible variations on the output bits.
• It must be emphasized that the linear attack exploits not only those bit positions at the input and the output when the linear relationship is often true, it also exploits those bit positions when such linear relationships are often false. The inequality case of the linear relationships of the sort shown above are more correctly referred to as **affine relationships**.

• As mentioned earlier, for an ideal SBox, all linear (and affine) relationships of the sort shown above will hold with a probability of 0.5. That is, if you feed the 256 possible different bit patterns into the input of a SBox, you should see such relationships to hold 128 times for all possible groupings of the input bit positions and the output bit positions. Any departure from this average is referred to as **linear approximation bias**. It is this bias that is exploited by a linear attack on a block cipher.

• So our first order of business is to characterize an SBox with regard to the prevalence of linear approximation biases. The two independent variables in a depiction of this bias are the input bit positions and the output bit positions. We can obviously express both with integers that range from 0 to 255, both ends inclusive. We will refer to these two integers as the **bit grouping integers**. The bits that are set in the bit-pattern representations of the two integers tell us which bit positions are involved in a linear (or an affine) relationship. For example, when the bit grouping integer for the input bits is 3 and the one for the output
bits is 12, we are talking about the following relationship:

\[ X_0 \otimes X_1 = Y_2 \otimes Y_3 \]

- By scanning through all 256 different possible bit patterns at the input to an SBox, and through the corresponding 256 different possible output bit patterns, we can count the number of times the equations of the type shown in Eq. (6) are satisfied. After we subtract the average value of 128 from these counts, we get what is referred to as the linear approximation table (LAT).

- What follows is a Perl script that can calculate LAT for two different choices of the SBox, depending on which of the two lines, (B4) or (B5), is left uncommented. The statement in line (B4) gives you an SBox that replaces a byte with its MI in \( GF(2^8) \) based on the AES modulus. On the other hand, the statement in line (B5) gives an SBox that is based on the lookup table defined in line (A8). The LAT itself is calculated in lines (B1) through (B26) of the script.

```perl
#!/usr/bin/perl
#
## linear_approximation_table_generator.pl
## Avi Kak (March 5, 2015)
## This script demonstrates how to generate the Linear Approximation Table that is
## needed for mounting a Linear Attack on a block cipher.
```

```bash
```
use strict;
use Algorithm::BitVector;
use Graphics::GnuplotIF;
$|++;

my $debug = 1;
my $AES_modulus = Algorithm::BitVector->new(bitstring => '100011011');  #(A1)

# Initialize LAT:
my $linear_approximation_table;  #(A2)
foreach my $i (0..255) {  #(A3)
    foreach my $j (0..255) {  #(A4)
        $linear_approximation_table->[$i][$j] = 0;  #(A5)
    }
}

# When SBox is based on lookup, we will use the "table" created by randomly
# permuting the number from 0 to 255:
#my $lookuptable = shuffle([0..255]);  #(A6)
#my @lookuptable = @$lookuptable;  #(A7)
my @lookuptable = qw(213 170 104 116 66 14 76 219 200 42 22 17 241 197 41 216 85 140
183 244 235 6 118 208 74 218 99 44 1 89 11 205 195 125 47 236 113
237 131 109 102 9 21 220 59 154 119 148 38 120 13 217 16 100 191 81
240 196 122 83 177 229 142 35 88 48 167 0 29 153 163 146 166 77 79
43 10 194 232 189 238 164 204 111 69 51 126 62 211 242 70 214 247 55
202 78 239 114 184 112 228 84 152 187 45 49 175 58 253 72 95 19 37
73 145 87 198 71 159 34 91 168 250 255 8 121 96 50 141 181 67 26 243
130 68 61 24 105 210 172 139 136 128 157 133 80 93 39 2 143 161 186 33
144 178 30 92 138 169 86 249 252 155 193 63 223 203 245 129 4 171
115 3 40 151 7 188 231 174 25 23 207 180 56 46 206 215 227 162 199
97 147 182 149 108 36 132 5 12 103 110 209 160 137 53 224 185 173
20 222 246 28 179 134 75 254 57 60 234 52 165 225 248 31 230 156
124 233 158 27 18 94 65 32 54 106 192 221 190 101 98 251 212 150
201 117 127 107 176 226 135 123 82 15 64 90);  #(A8)

gen_linear_approximation_table();  #(A9)

# The call shown below will show that part of the LAT for which both the input and
# the output bit grouping integers are in the range (0, 32):
display_portion_of_LAT(0, 32);  #(A10)

# This call makes a graphical plot for a portion of the LAT. The bit grouping index
# ranges for both the input and the output bytes are 32 to 64:
plot_portion_of_LAT($linear_approximation_table, 32, 64);  #(A11)
#
## The following call makes a hardcopy of the plot:
plot_portion_of_LAT($linear_approximation_table, 32, 64, 3);  #(A12)

# You have two choices for the SBox in lines (B4) and (B5). The one is line (B4) is
# uses MI in GF(2^8) based on the AES modulus. And the one in line (B5) uses the
# lookup table defined above in line (A8). Comment out the one you do not want.
sub gen_linear_approximation_table {  #(B1)
    foreach my $x (0 .. 255) {  # specify a byte for the input to the SBox
        print "\input byte = $x\n" if $debug;  #(B2)
my $a = Algorithm::BitVector->new(intVal => $x, size => 8);  # (B3)
# Now get the output byte for the SBox:
my $c = get_sbox_output_MI($a);  # (B4)
my $c = get_sbox_output_lookup($a);  # (B5)
my $y = int($c);  # (B6)
foreach my $bit_group_from_x (0 .. 255) {  # (B7)
  my @input_bit_positions;  # (B8)
  foreach my $pos (0..7) {  # (B9)
    push @input_bit_positions, $pos if ($bit_group_from_x >> $pos) & 1;  # (B10)
  }  # (B11)
  my $input_linear_sum = 0;  # (B12)
  foreach my $pos (@input_bit_positions) {  # (B13)
    $input_linear_sum ^= (($x >> $pos) & 1);  # (B14)
  }
  foreach my $bit_group_from_y (0 .. 255) {  # (B15)
    my @output_bit_positions;  # (B16)
    foreach my $pos (0..7) {  # (B17)
      push @output_bit_positions, $pos if ($bit_group_from_y >> $pos) & 1;  # (B18)
    }
    my $output_linear_sum = 0;  # (B19)
    foreach my $pos (@output_bit_positions) {  # (B20)
      $output_linear_sum ^= (($y >> $pos) & 1);  # (B21)
    }
    $linear_approximation_table->[\$bit_group_from_x][\$bit_group_from_y]++  # (B22)
    if $input_linear_sum == $output_linear_sum;  # (B23)
  }  # (B24)
  $linear_approximation_table->[\$i][\$j] -= 128;  # (B25)
}  # (B26)

sub get_sbox_output_MI {  # (C1)
  my $in = shift;  # (C2)
  return int($in) != 0 ? $in->gf_MI($AES_modulus, 8) :  # (C3)
    Algorithm::BitVector->new(intVal => 0);  # (C4)
}

sub get_sbox_output_lookup {  # (D1)
  my $in = shift;  # (D2)
  return Algorithm::BitVector->new(intVal => $lookuptable[int($in)], size => 8);  # (D3)
}

# Fisher-Yates shuffle:
sub shuffle {  # (E1)
  my $arr_ref = shift;  # (E2)
  my $i = @$arr_ref;  # (E3)
  while ( $i-- ) {  # (E4)
    my $j = int rand( $i + 1 );  # (E5)
    @$arr_ref[ $i, $j ] = @$arr_ref[ $j, $i ];  # (E6)
  }
  return $arr_ref;  # (E8)
sub display_portion_of_LAT {  
    my $lower = shift;  
    my $upper = shift;  
    foreach my $i ($lower .. $upper - 1) {  
        print "\n";  
        foreach my $j ($lower .. $upper - 1) {  
            print "$linear_approximation_table->[\$i][\$j] \n";  
        }  
    }  
}

sub plot_portion_of_LAT {  
    my $hist = shift;  
    my $lower = shift;  
    my $upper = shift;  
    my $pause_time = shift;  
    my @plot_points = ();  
    my $bin_width = my $bin_height = 1.0;  
    my ($x_min, $y_min, $x_max, $y_max) = ($lower, $lower, $upper, $upper);  
    foreach my $y ($y_min .. $y_max - 1) {  
        foreach my $x ($x_min .. $x_max - 1) {  
            push @plot_points, [$x, $y, $hist->[\$y][\$x]];  
        }  
    }  
    @plot_points = sort {$a->[0] <=> $b->[0]} @plot_points;  
    @plot_points = sort {$a->[1] <=> $b->[1] if $a->[0] == $b->[0]} @plot_points;  
    my $temp_file = "__temp.dat";  
    open OUTFILE, ">$temp_file" or die "Cannot open temporary file: $!";  
    my ($first, $oldfirst);  
    $oldfirst = $plot_points[0]->[0];  
    foreach my $sample (@plot_points) {  
        $first = $sample->[0];  
        if ($first == $oldfirst) {  
            my @out_sample;  
            $out_sample[0] = $sample->[0];  
            $out_sample[1] = $sample->[1];  
            $out_sample[2] = $sample->[2];  
            print OUTFILE "@out_sample\n";  
        } else {  
            print OUTFILE "\n";  
        }  
    }  
}

# Displays in your terminal window the bin counts (minus 128) in the LAT calculated  
# in lines (B1) through (B26). You can control the portion of the display by using  
# the first argument to set the lower bin index and the second argument the upper  
# bin index along both dimensions. Therefore, what you see is always a square  
# portion of the LAT.

# Displays with a 3-dimensional plot a square portion of the LAT. Along both the X  
# and the Y directions, the lower bound on the bin index is supplied by the SECOND  
# argument and the upper bound by the THIRD argument. The last argument is needed  
# only if you want to make a hardcopy of the plot. The last argument is set to the  
# number of second the plot will be flashed in the terminal screen before it is  
# dumped into a '.png' file.

• For the case when you run script with the SBox based on MI calculations in $GF(2^8)$, shown below is a small portion of the LAT constructed by the script. [The portion of the LAT shown below was dictated by the page width constraints.] In keeping with the explanation provided earlier, you can see that the topmost row and the leftmost column values are all zero, as we expect them to be. The entries at the other locations tell us how much positive and negative bias there exists in the linear relationships corresponding to those cells. Looking at the seventh entry (of column index 6) in the second row (of row index 1), we can say that the relationship $X_1 \otimes X_2 \otimes Y_1 = 0$ is true with a probability of $12/256$, and so on. Note that the full table that is calculated by the Perl script is $256 \times 256$. Theory dictates that the sum of the entries in each row or each column
must be either 128 or -128.

\[
\begin{array}{cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccco
\end{array}
\]

- Shown in Figure 8.7 is a plot of a portion of the LAT that was calculated by the Perl script for the case of SBox based on MI in $GF(2^8)$. The portion shown is a 32 × 32 portion of the table starting at the cell located at (32, 32).

- If you comment out line (B3) and uncomment line (B4) so that
the SBox would be based on the lookup table in line (A8), the portion of the plot shown in Figure 8.7 becomes what is shown
Figure 7: Shown is a portion of the LAT for an SBox that calculates MIs in $GF(2^8)$ using the AES modulus. (This figure is from Lecture 8 of “Computer and Network Security” by Avi Kak.)

in Figure 8.8. Note that the largest peaks in the LAT of Figure 8.8 are larger than the largest peaks in the LAT of Figure 8.7. That implies that the SBox based on the lookup table of line (A8) results in larger biases for some of the linear equations compared to the SBox that is based on MI in $GF(2^8)$.

- Representing an arbitrary linear form $x_{i_1} \otimes \ldots \otimes x_{i_m} \otimes y_{j_1} \otimes \ldots \otimes y_{j_n}$ by $\zeta$, the cell values in a LAT allow us to write down the following probabilities: $\text{prob}(\zeta = 0) = p$ and $\text{prob}(\zeta = 1) = 1 - p$.

- An important part of the formulation of the linear attack is the
use of Matsui’s *piling-up lemma* to estimate the joint probabilities $\text{prob}(\zeta_1, \zeta_2, \ldots) = 0$ and $\text{prob}(\zeta_1, \zeta_2, \ldots) = 1$ with each $\zeta_i$ expressing one of the linear forms for the $i^{th}$ round.

- After constructing a LAT for the SBoxes used in a cipher and after estimating the join probabilities associated with the linear equations over multiple rounds, executing a linear attack involves the following steps: (1) You string together the linear forms of the type shown earlier across the rounds but not including the last round and estimate the probabilistic biases associated with the linear forms (these can also be affine forms). (2) Considering different possible candidate keys for the last round, you partially
decrypt the ciphertext. For each candidate key for the last round, this gives you candidate output bits for the last-round Sbox. (3) Using these candidate output bits for the last round, you accumulate votes for the different candidates for the last-round key depending on the extent to which candidate SBox output bits for the last round are consistent with the linear forms constructed from the first $n - 1$ rounds.

- To make the above explanation more specific, assume that the block size in our cipher is just one byte and that there are no permutations involved. [See the previously mentioned tutorial by Howard Heys for a more realistic example that involves both substitutions and permutations.] Let $P_i$ denote the $i^{th}$ bit of the plaintext byte entering the first round. We will assume that each round consists of a byte substitution by the SBox, followed by the addition of the round key. In general, we will use $X_{r,i}$ to denote the $i^{th}$ input bit to the $r^{th}$ round and let $Y_{r,j}$ denote the $j^{th}$ output bit of the SBox in the same round. Additionally, let $K_{r,k}$ denote the $k^{th}$ bit of the round key for the $r^{th}$ round. We can now construct linear relationships of the following sort that span all of the rounds together:

$$P_{i_1} \otimes P_{i_2} \otimes \ldots \otimes Y_{1,j_1} \otimes \ldots Y_{1,j_m} \otimes \ldots K_1 \otimes Y_{2,j_1} \otimes \ldots Y_{2,j_1} \otimes \ldots \ldots K_2 \otimes Y_{3,j_1} \otimes \ldots Y_{3,j_1} \otimes \ldots \ldots Y_{n-1,j_1} \otimes \ldots Y_{n-1,j_m}$$

where we have used the fact that the output of each SBox, after the addition of the round key, becomes the input to the next round. That is, $Y_{r,i} \otimes K_{r,i}$ becomes $X_{r+1,i}$. The above linear
form may be expressed in the following form:

\[
\text{XOR sum of only } P \text{ and } Y \text{ variables} \quad \otimes \quad \text{XOR sum of key bits in rounds from 1 through } n - 1 = 0
\]

which can be abbreviated to

\[
\text{XOR sum of only } P \text{ and } Y \text{ variables} \quad \otimes \quad \Sigma_K = 0
\]

where \( \Sigma_K \) is the linear form that involves only the key bits from the first \( n - 1 \) rounds.

- We have only two possibilities for \( \Sigma_K \). Either it is equal to 0 or to 1. If we assume that both are equiprobable, that eliminates the influence of \( \Sigma_K \) on the bias associated with the rest of the linear equation shown above. Subsequently, it becomes easy to decide how much weight to give to a candidate key for the last round depending on the probability associated with the linear form that depends only on the inputs and the outputs of the Sboxes.

- That brings us to the **interpolation attack**. The interpolation attack seeks to model the behavior of an SBox with a polynomial in \( GF(2^8) \). Recall that the SBox is the only source of nonlinearity in transforming plaintext into ciphertext. (All of the permutation operations are obviously linear.) We also recognize that what an SBox does must be invertible on a one-one basis (in other words, the input/output mapping provided by an SBox must be bijective).
Let’s say that it is possible to represent the round operation that involves an SBox calculation following by key mixing by the algebraic function $f_i(c_{i-1}, K_i)$ where $c_{i-1}$ is the input to the round and $K_i$ is the round key. Let’s further say that $f_i$ can be expressed as a polynomial in $GF(2^8)$ over the input to the round and that the unknown $K_i$ values can be expressed as the coefficients of this polynomial. It was shown by Jakobsen and Knudsen that when such a polynomial is of low degree, its coefficients can be estimated from a set of plaintext-ciphertext pairs. Subsequently, an attacker would be able to invert the polynomial to find the plaintext for a given ciphertext without having to know the encryption key used.
8.10: HOMEWORK PROBLEMS

1. With regard to the first step of processing in each round of AES on the encryption side: How does one look up the $16 \times 16$ S-box table for byte-by-byte substitutions? In other words, assuming I want a substitute byte for the byte $b_7b_6b_5b_4b_3b_2b_1b_0$, where each $b_i$ is a single bit, how do I use these bits to find the replacement byte in the S-box table?

2. What are the steps that go into the construction of the $16 \times 16$ S-box lookup table?

3. 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?

4. The second step in each round permutes the bytes in each row of the state array. What is the permutation formula that is used?
5. Describe the “mix columns” transformation that constitutes the third step in each round of AES.

6. 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?

7. 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 $w_4$, $w_5$, $w_6$, $w_7$?

8. Going back to the previous question, the formula that yields $w_4$ is

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

What goes into computing $g()$?

9. **Programming Assignment:**

Write a Perl or Python based implementation of AES. As you know, each round of processing involves the following four steps:

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

Your implementation must include the code for creating the two 16 × 16 tables that you need for the byte substitution steps, one for encryption and the other for decryption. Note that the lookup table you construct for encryption is also used in the key expansion algorithm.

The effort that it takes to do this homework is significantly reduced if you use the BitVector module in Python and the Algorithm::BitVector module in Perl. The following method of in these modules should be particularly useful for constructing the two lookup tables for byte substitutions:

\texttt{gf\_MI}

This method returns the multiplicative inverse of a bit pattern in $GF(2^n)$ with respect to a modulus bit pattern that corresponds to the irreducible polynomial used. To illustrate with Python the sort of call you’d need to make, the API documentation for the BitVector module shows the following example code on how to call this method:

```python
modulus = BitVector(bitstring = '100011011')
n = 8
a = BitVector(bitstring = '00110011')
multiplicative_inverse = a.gf_MI(modulus, n)
print multiplicative_inverse # 01101100
```

Note that the variable \texttt{modulus} is set to the BitVector that corresponds to the AES irreducible polynomial. The variable \texttt{a} can be
set to any arbitrary \texttt{BitVector} whose multiplicative inverse you are interested in.

The other \texttt{BitVector} method that should prove particularly useful for this homework is:

\texttt{gf\_multiply\_modular}

This method lets you multiply two bit patterns in $GF(2^n)$. To multiply two bit patterns $a$ and $b$, both instances of the \texttt{BitVector} class, when the modulus bit pattern is $mod$, you invoke

\[ a.gf\_multiply\_modular(b, mod, n) \]

where $n$ is the exponent for 2 in $GF(2^n)$. For this homework problem, $n$ is obviously 8.

Your implementation should be for 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.