Lecture 8: AES: The Advanced Encryption Standard

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

February 3, 2022

3:52pm

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.
- Python and Perl implementations for creating the lookup tables for the byte substitution steps in encryption and decryption.
- Python and Perl implementations of the Key Expansion Algorithms for the 128 bit, 192 bit, and 256 bit AES.
- Perl implementations for creating histograms of the differentials and for constructing linear approximation tables in attacks on block ciphers.

CONTENTS

Back to TOC

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 use of "row" and "column" in the previous bullet, you need to think of the input 128-bit block as consisting of a 4×4 array of bytes, arranged as follows:

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

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

- The 4×4 array 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, very similar steps

are used in encryption and decryption, their implementations are not identical and the order in which the steps are invoked 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 chosen by NIST as a potential replacement 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 always-in-the-news cryptographer Bruce Schneier. Rijndael was selected from 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.

- Like DES, AES is an *iterated block cipher* in which plaintext is subject to multiple rounds of processing, with each round applying the same overall transformation function to the incoming block. [When we say that each round applies the same transformation function to the incoming block, that similarity is at the functional level. However, the implementation of the transformation function in each round involves a key that is specific to that round — this key is known as the round key. Round keys are derived from the user-supplied encryption key.]

- Unlike DES, AES is an example of *key-alternating block ciphers*. In such ciphers, each round first applies a diffusion-achieving transformation operation — which may be a combination of linear and nonlinear steps — to the entire incoming block, which is then followed by the application of the round key to the entire block. As you'll recall, DES is based on the Feistel structure in which, for each round, one-half of the block passes through unchanged and the other half goes through a transformation that depends on the S-boxes and the round key. Key alternating ciphers lend themselves well to theoretical analysis of the security of the ciphers.

- For another point of contrast between DES and AES, whereas

DES is a bit-oriented cipher, AES is a byte-oriented cipher. [Remember, how in DES we segmented the right-half 32 bits of the incoming 64-bit block into eight segments of 4-bits each. And how we prepended each 4-bit segment with the last bit of the previous 4-bit segment and appended to each 4-bit segment the first bit of the next 4-bit segment. Subsequently, in order to find the substitution 4-bits for an incoming 4-bit segment, we used the first and the last bit thus acquired for indexing into the four rows of a 4×16 S-box, while using the 4-bit segment itself for indexing into the columns of the S-Box.] The substitution step in DES requires bit-level access to the block coming into a round. On the other hand, all operations in AES are purely byte-level, which makes for convenient and fast software implementation of AES.

- About the security of AES, considering how many years have passed since the cipher was introduced in 2001, all of the threats against the cipher remain theoretical — meaning that their time complexity is way beyond what any computer system will be able to handle for a long time to come. [As you know, for the 128-bit key AES, the worst-case time complexity for a brute-force attack would be 2^{128}. Such a brute-force attack would be considered to be an example of a theoretical attack since it is beyond the realm of any practical implementation. There is a meet-in-the-middle attack called the *biclique attack* that very marginally improves upon this time complexity to around 2^{126} — which is still just a theoretical attack. The biclique attack was presented by Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger in their 2011 publication "Biclique Cryptanalysis of the Full AES".]

- AES was designed using the *wide-trail strategy*. As described in the publication "*Security of a Wide Trail Design*" by Joan Daemen and Vincent Rijmen, wide-trail design for a block cipher involves: (1) A local nonlinear transformation (as

supplied by the substitution step in AES); and (2) A linear mixing transformation that provides high diffusion. The phrase "wide trail" refers to dispersal of the probabilities that one can associate with the bits at certain specific positions in a bit block as it propagates through the rounds.

- If you are seriously interested in the algebraic foundations of AES and also of the attacks that are being attempted on the cipher, I'd recommend the book "*Algebraic Aspects of the Advanced Encryption Standard*," by Carlos Cid, Sean Murphy, and Matthew Robshaw. This book was originally published by Springer, but is now available for free download on the web. Just Google it.

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

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

Back to TOC

8.2 THE ENCRYPTION KEY AND ITS EXPANSION

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

- The four column words of the key array 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. Of these, the first four words are used for adding to the input state array before any round-based processing can begin, and the remaining 40 words used for the ten rounds of processing that are required for the case a 128-bit encryption key.

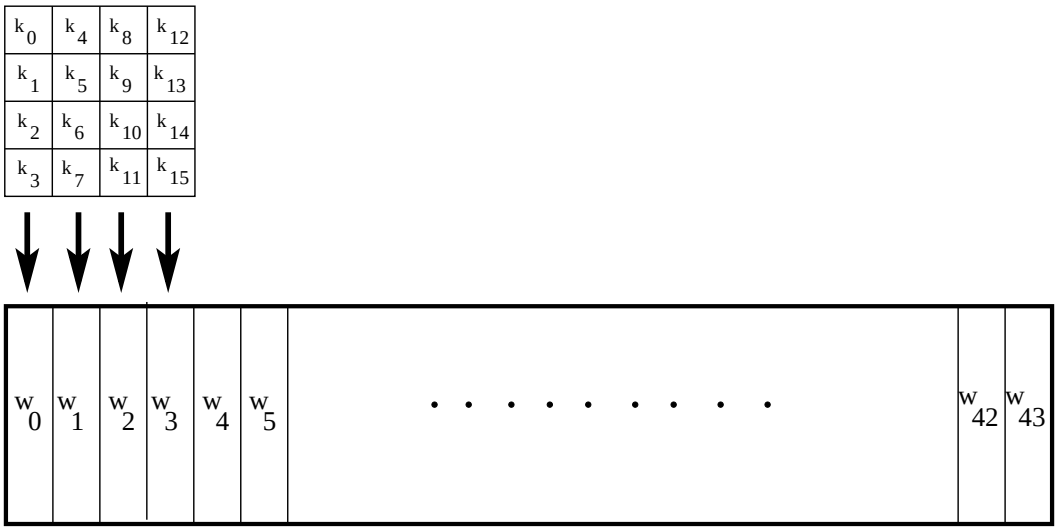

Figure 1: *This figure shows the four words of the original 128-bit key being expanded into a key schedule consisting of 44 words. Section 8.8 explains the procedure used for this key expansion.* (*This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak*)

Back to TOC

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.

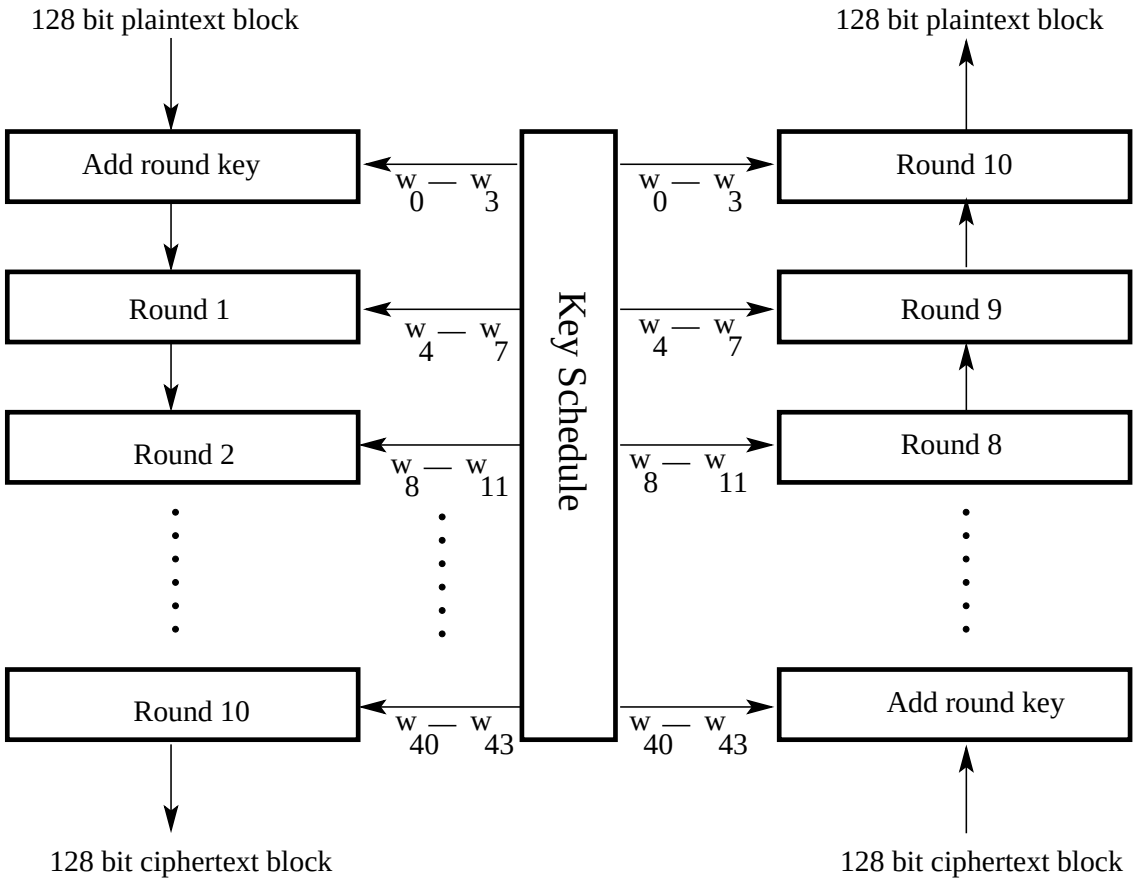

Figure 2: *The overall structure of AES for the case of 128-bit encryption key.* (*This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak*)

Back to TOC

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. [See the end of the previous section as to what steps are not allowed in the last round.]

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

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

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

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 during decryption is denoted **InvShiftRows** for Inverse Shift-Row Transformation.)

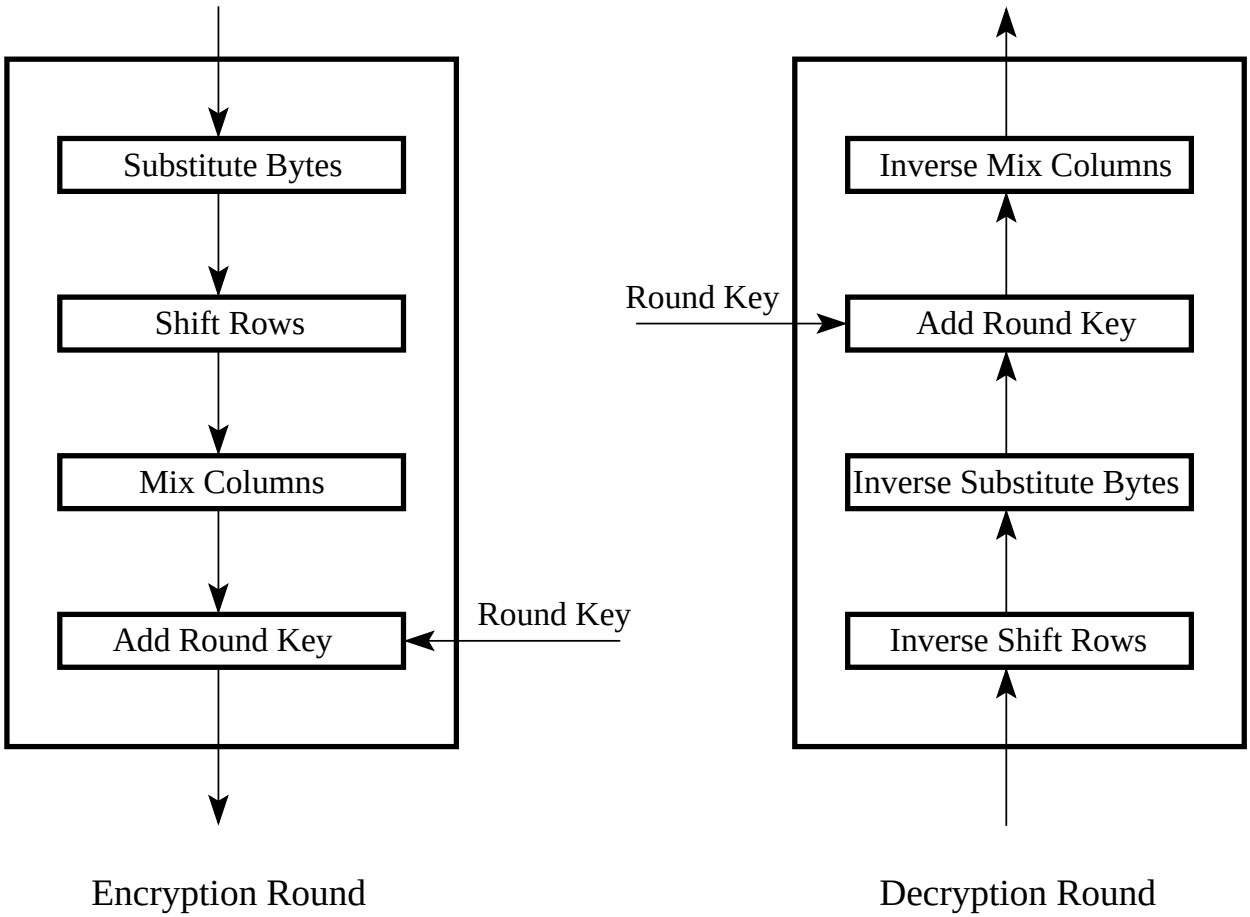

Figure 3: *One round of encryption is shown at left and one round of decryption at right.* (*This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak*)

- 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.)

Back to TOC

8.5 THE SUBSTITUTE BYTES STEP: SubBytes and InvSubBytes

- This is a byte-by-byte substitution using a rule that stays the same in **all** encryption rounds. The byte-by-byte substitution rule is different for the decryption chain, but again it stays the same for all the rounds.

- The presentation in the rest of this section is organized as follows:

  – **The modern way** of explaining the byte substitution step that allows us to find the substitute byte for a given byte by simply looking up a pre-computed 256-element array of numbers.

  – **The traditional way** of explaining the byte substitution step that involves using a 16×16 lookup table.

  – Perl and Python implementations of the byte substitution step. **These implementations are based on the modern explanation of the step.** (Obviously, as you would expect, both explanations lead to the same final answer for byte substitution.)

- **In the modern way** of explaining the byte substitution step for the encryption chain, let $\mathbf{x}_{in}$ be a byte of the state array for which we seek a substitute byte $\mathbf{x}_{out}$. We can write $\mathbf{x}_{out} = f(\mathbf{x}_{in})$. The function $f()$ involves two nonlinear operations: (i) We first find the multiplicative inverse $\mathbf{x}' = \mathbf{x}_{in}{}^{-1}$ in $GF(2^8)$; **and** (ii) then we scramble the bits of $\mathbf{x}'$ by XORing $\mathbf{x}'$ with four different circularly rotated versions of itself and with a special constant byte $\mathbf{c} = 0x63$. The four circular rotations are through 4, 5, 6, and 7 bit positions to the right. As you will see later in this section, this bit scrambling step can be expressed by the relation: $\mathbf{x}_{out} = A \cdot \mathbf{x}' + \mathbf{c}$.

- When using my BitVector module, the byte substitution step as explained above can be implemented with just a couple of calls to the module functions. The first operation of the step, which involves calculating the multiplicative inverse of a byte $\mathbf{x}$ in $GF(2^8)$, can be carried out by invoking the function `gf_MI()` on the `BitVector` representation of $\mathbf{x}$. The second operation that requires XORing a byte with circularly shifted versions of itself is even more trivial, as you will see in the Perl and Python code shown later in this section.

- The modern explanation of the byte substitution step as presented above applies equally well to the decryption chain, **except for the fact that you first apply the bit scrambling operation to the byte and then you find its multiplicative inverse in $GF(2^8)$.**

- **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.**

- **I'll now present the more traditional explanation of the byte substitution step.** As mentioned earlier, it involves using a 16×16 table. To find the substitute byte for a given input byte, we divide the input byte into two 4-bit patterns, each yielding an integer value between 0 and 15. (We can represent these by their hex values 0 through F.) One of the hex values is used as a row index and the other as a column index for reaching into the 16×16 lookup table.

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

Back to TOC

8.5.1 Traditional Explanation of Byte Substitution: Constructing the 16×16 Lookup Table

- We first fill each cell of the 16×16 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

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

- We next replace the value in each cell by its multiplicative inverse in $GF(2^8)$ based on the irreducible polynomial

$x^8 + x^4 + x^3 + x + 1$. **The 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 `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 **bit** b_i of the byte stored in a cell of the lookup table:

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

where c_i is the i^{th} bit of a specially designated byte c whose hex

value is 0x63. ($c_7c_6c_5c_4c_3c_2c_1c_0 \equiv 01100011$)

- The above bit-scrambling step is better visualized as the following matrix-vector operation. Note that all of the additions in the product of the matrix and the vector are actually XOR operations. That that should be so is made evident by comparing the matrix-vector product form shown below with the last equation on the previous page. [Because of the $[A]\vec{x} + \vec{b}$ appearance of this transformation, it is commonly referred to as the affine transformation.]

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

- **The very important role played by the c byte of value 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 16×16 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 Section 4.6 of 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. ($d_7 d_6 d_5 d_4 d_3 d_2 d_1 d d c_0 \;=\; 00000101$) Finally, you replace the byte in the cell by its multiplicative inverse in $GF(2^8)$. [**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 $S\_box(a) \;=\; a$ for any a. Neither do we want $S\_box(a) \;=\; \bar{a}$ where $\bar{a}$ is the bitwise complement of a.

Back to TOC

8.5.2 Python and Perl Implementations for the AES Byte Substitution Step

- Section 8.5 and the Subsection 8.5.1 presented two different ways of implementing the AES byte substitution step. As stated earlier in Section 8.5, both these explanations are equivalent — in the sense that either will result in the same substitution byte for a given input byte.

- This subsection shows my Python and Perl implementations of the more modern explanation of byte substitution described in Section 8.5. You will be surprised how easy it is to write this code if you are using my `BitVector` module in Python and the `Algorithm::BitVector` module in Perl.

- What follows is a Python implementation of the explanation. The goal of the `for` loop is to construct a 256 element array of lookup values for integers ranging from 0 through 255. For each integer in the range 0 through 255, we first find its multiplicative inverse in $GF(2^8)$, then we XOR the result with four different circularly rotated versions of the result, and also XOR the result with the constant $\mathbf{c} = 0x63$. We do the same thing for the decryption lookup array, except that we first do the XORing and then we compute the multiplicative inverse.

```
#!/usr/bin/env python

##  gen_tables.py
##  Avi Kak (February 15, 2015)

##  This is a Python implementation of the byte substitution explanations in Sections
##  8.5 and 8.5.1 of Lecture 8.  In keeping with the explanation in Section 8.5, the
##  goal here is to construct two 256-element arrays for byte substitution, one for
##  the SubBytes step that goes into the encryption rounds of the AES algorithm, and
##  the other for the InvSubBytes step that goes into the decryption rounds.

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/env perl

##  gen_tables.pl
##  Avi Kak (February 16, 2015)

##  This is a Perl implementation of the byte substitution explanations in Sections
##  8.5 and 8.5.1 of Lecture 8.  In keeping with the explanation in Section 8.5, the
##  goal here is to construct two 256-element arrays for byte substitution, one for
##  the SubBytes step that goes into the encryption rounds of the AES algorithm, and
##  the other for the InvSubBytes step that goes into the decryption rounds.

use strict;
use warnings;
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:\n";
print "@subBytesTable\n";
print "\nSBox for Decryption:\n";
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)

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

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

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

The Python and Perl scripts in this section can be downloaded from the link associated with Lecture 8 at the "Lecture Notes" website.

Now ask yourself the following questions:

1. Why is the value decimal 99 in the upper-left corner of the encryption look-up "table" presented in the upper half of the previous page?

2. Why is the value decimal 0 in the 7^{th} row and the 4^{th} column of the decryption lookup table in the lower half of the previous page?

Back to TOC

8.6 THE SHIFT ROWS STEP: ShiftRows and InvShiftRows

- This is where the array 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

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

- 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.

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

Back to TOC

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

$$s'_{0,j} \quad = \quad (\mathbf{0x02} \times s_{0,j}) \;\otimes\; (\mathbf{0x03} \times s_{1,j}) \;\otimes\; s_{2,j} \;\otimes\; s_{3,j}$$

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

$$s'_{1,j} \quad = \quad s_{0,j} \;\otimes\; (0\mathtt{x}02 \times s_{1,j}) \;\otimes\; (0\mathtt{x}03 \times s_{2,j}) \;\otimes\; s_{3,j}$$

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

$$s'_{2,j} \quad = \quad s_{0,j} \;\otimes\; s_{1,j} \;\otimes\; (0\mathtt{x}02 \times s_{2,j}) \;\otimes\; (0\mathtt{x}03 \times s_{3,j})$$

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

$$s'_{3,j} \quad = \quad (0\mathtt{x}03 \times s_{0,j}) \;\otimes\; s_{1,j} \;\otimes\; s_{2,j} \;\otimes\; (0\mathtt{x}02 \times s_{3,j})$$

- 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

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

Back to TOC

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 4×4 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

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

- 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 $w_0, w_1, w_2, w_3, w_4, w_5, ..., w_{43}$?

- 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.

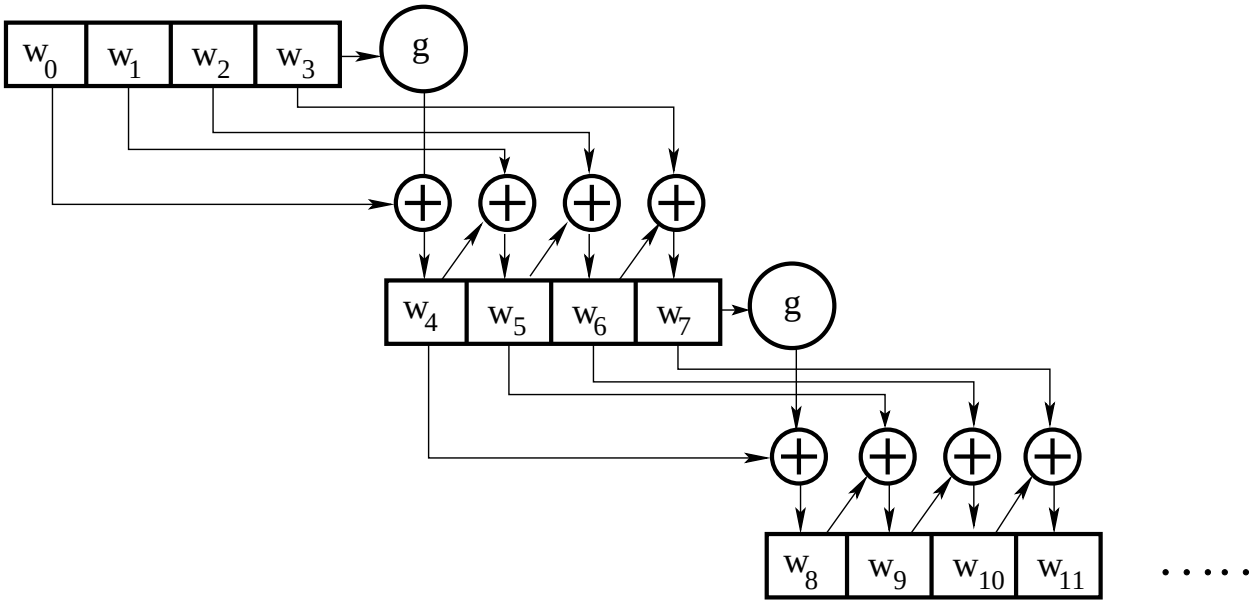

Figure 4: *The key expansion takes place on a four-word to four-word basis as shown here.* (*This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak*)

Back to TOC

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

$$w_{i+5} = w_{i+4} \otimes w_{i+1} \quad (1)$$
$$w_{i+6} = w_{i+5} \otimes w_{i+2} \quad (2)$$
$$w_{i+7} = w_{i+6} \otimes w_{i+3} \quad (3)$$

  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:

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

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

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

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

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

the **SubBytes** step of the encryption rounds. [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.

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

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

$$\begin{aligned} RC[1] \quad &= \quad \mathtt{0x}01 \\ RC[j] \quad &= \quad \mathtt{0x}02 \times RC[j-1] \end{aligned}$$

Recall from Lecture 7 that multiplication by $\mathtt{0x}02$ 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. [Consider what happens when the key length is 192 bits: Since the round-based processing and the size of the input block remain the same as described earlier in this lecture, each round will still use only 4 words of the key schedule. Just as we organized the 128-bit key in the form of 4 key words for the purpose of key expansion, we organize the 192 bit key in the form of **six** words. The key expansion algorithm will take us from six words to six words — for a total of **nine** key-expansion steps — with each step looking the same as what we described at the beginning of this section. Yes, it is true that the key expansion will now generate a total of 54 words while we need only 52 — we simply ignore the last two words of the key schedule. With regard to the details of going from the six words of the 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.)

Back to TOC

8.8.2 Python and Perl Implementations of the Key Expansion Algorithm

- In this section, I'll first present a Python implementation of the key expansion algorithm described in the previous subsection. That will be followed by a Perl implementation of the same.

- With regard to key expansion, the main focus of the previous subsection was the 128-bit AES. Toward the end of the previous subsection, I briefly described the modifications needed for the case of 192-bit and 256-bit AES. The goal of the implementation shown in this section is to clarify the various steps for all three cases.

- When you execute the Python code shown below, it will prompt you for AES key size — obviously, the number you enter must be one of 128, 192, and 256.

- Subsequently, it will prompt you for the key. You are allowed to enter any number of characters for the key. If the length of the key you enter is shorter than what is needed to fill the full width of the AES key size, the script appends the character '0' to your key to bring it up to the required size. On the other hand, if you enter a key longer than what is needed, it will only use the

number of characters it needs.

```
#!/usr/bin/env python

##  gen_key_schedule.py
##  Avi Kak  (April 10, 2016; bug fix: January 27, 2017; doc errors fixed: February 2, 2018)

##  This script is for demonstrating the AES algorithm for generating the
##  key schedule.

##  It will prompt you for the key size, which must be one of 128, 192, 256.

##  It will also prompt you for a key.  If the key you enter is shorter
##  than what is needed for the AES key size, we add zeros on the right of
##  the key so that its length is as needed by the AES key size.

import sys
from BitVector import *

AES_modulus = BitVector(bitstring='100011011')

def main():
    key_words = []
    keysize, key_bv = get_key_from_user()
    if keysize == 128:
        key_words = gen_key_schedule_128(key_bv)
    elif keysize == 192:
        key_words = gen_key_schedule_192(key_bv)
    elif keysize == 256:
        key_words = gen_key_schedule_256(key_bv)
    else:
        sys.exit("wrong keysize --- aborting")
    key_schedule = []
    print("\nEach 32-bit word of the key schedule is shown as a sequence of 4 one-byte integers:")
    for word_index,word in enumerate(key_words):
        keyword_in_ints = []
        for i in range(4):
            keyword_in_ints.append(word[i*8:i*8+8].intValue())
        if word_index % 4 == 0: print("\n")
        print("word %d:  %s" % (word_index, str(keyword_in_ints)))
        key_schedule.append(keyword_in_ints)
    num_rounds = None
    if keysize == 128: num_rounds = 10
    if keysize == 192: num_rounds = 12
    if keysize == 256: num_rounds = 14
    round_keys = [None for i in range(num_rounds+1)]
    for i in range(num_rounds+1):
        round_keys[i] = (key_words[i*4] + key_words[i*4+1] + key_words[i*4+2] +
                                                    key_words[i*4+3]).get_bitvector_in_hex()
    print("\n\nRound keys in hex (first key for input block):\n")
    for round_key in round_keys:
        print(round_key)

def gee(keyword, round_constant, byte_sub_table):
    '''
```

```
    This is the g() function you see in Figure 4 of Lecture 8.
    '''
    rotated_word = keyword.deep_copy()
    rotated_word << 8
    newword = BitVector(size = 0)
    for i in range(4):
        newword += BitVector(intVal = byte_sub_table[rotated_word[8*i:8*i+8].intValue()], size = 8)
    newword[:8] ^= round_constant
    round_constant = round_constant.gf_multiply_modular(BitVector(intVal = 0x02), AES_modulus, 8)
    return newword, round_constant

def gen_key_schedule_128(key_bv):
    byte_sub_table = gen_subbytes_table()
    #  We need 44 keywords in the key schedule for 128 bit AES.  Each keyword is 32-bits
    #  wide. The 128-bit AES uses the first four keywords to xor the input block with.
    #  Subsequently, each of the 10 rounds uses 4 keywords from the key schedule. We will
    #  store all 44 keywords in the following list:
    key_words = [None for i in range(44)]
    round_constant = BitVector(intVal = 0x01, size=8)
    for i in range(4):
        key_words[i] = key_bv[i*32 : i*32 + 32]
    for i in range(4,44):
        if i%4 == 0:
            kwd, round_constant = gee(key_words[i-1], round_constant, byte_sub_table)
            key_words[i] = key_words[i-4] ^ kwd
        else:
            key_words[i] = key_words[i-4] ^ key_words[i-1]
    return key_words

def gen_key_schedule_192(key_bv):
    byte_sub_table = gen_subbytes_table()
    #  We need 52 keywords (each keyword consists of 32 bits) in the key schedule for
    #  192 bit AES.  The 192-bit AES uses the first four keywords to xor the input
    #  block with.  Subsequently, each of the 12 rounds uses 4 keywords from the key
    #  schedule. We will store all 52 keywords in the following list:
    key_words = [None for i in range(52)]
    round_constant = BitVector(intVal = 0x01, size=8)
    for i in range(6):
        key_words[i] = key_bv[i*32 : i*32 + 32]
    for i in range(6,52):
        if i%6 == 0:
            kwd, round_constant = gee(key_words[i-1], round_constant, byte_sub_table)
            key_words[i] = key_words[i-6] ^ kwd
        else:
            key_words[i] = key_words[i-6] ^ key_words[i-1]
    return key_words

def gen_key_schedule_256(key_bv):
    byte_sub_table = gen_subbytes_table()
    #  We need 60 keywords (each keyword consists of 32 bits) in the key schedule for
    #  256 bit AES. The 256-bit AES uses the first four keywords to xor the input
    #  block with.  Subsequently, each of the 14 rounds uses 4 keywords from the key
    #  schedule. We will store all 60 keywords in the following list:
    key_words = [None for i in range(60)]
    round_constant = BitVector(intVal = 0x01, size=8)
```

```
    for i in range(8):
        key_words[i] = key_bv[i*32 : i*32 + 32]
    for i in range(8,60):
        if i%8 == 0:
            kwd, round_constant = gee(key_words[i-1], round_constant, byte_sub_table)
            key_words[i] = key_words[i-8] ^ kwd
        elif (i - (i//8)*8) < 4:
            key_words[i] = key_words[i-8] ^ key_words[i-1]
        elif (i - (i//8)*8) == 4:
            key_words[i] = BitVector(size = 0)
            for j in range(4):
                key_words[i] += BitVector(intVal =
                                byte_sub_table[key_words[i-1][8*j:8*j+8].intValue()], size = 8)
            key_words[i] ^= key_words[i-8]
        elif ((i - (i//8)*8) > 4) and ((i - (i//8)*8) < 8):
            key_words[i] = key_words[i-8] ^ key_words[i-1]
        else:
            sys.exit("error in key scheduling algo for i = %d" % i)
    return key_words

def gen_subbytes_table():
    subBytesTable = []
    c = BitVector(bitstring='01100011')
    for i in range(0, 256):
        a = BitVector(intVal = i, size=8).gf_MI(AES_modulus, 8) if i != 0 else BitVector(intVal=0)
        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))
    return subBytesTable

def get_key_from_user():
    key = keysize = None
    if sys.version_info[0] == 3:
        keysize = int(input("\nAES Key size:  "))
        assert any(x == keysize for x in [128,192,256]), \
                                    "keysize is wrong (must be one of 128, 192, or 256) --- aborting"
        key = input("\nEnter key (any number of chars):  ")
    else:
        keysize = int(raw_input("\nAES Key size:  "))
        assert any(x == keysize for x in [128,192,256]), \
                                    "keysize is wrong (must be one of 128, 192, or 256) --- aborting"
        key = raw_input("\nEnter key (any number of chars):  ")
    key = key.strip()
    key += '0' * (keysize//8 - len(key)) if len(key) < keysize//8 else key[:keysize//8]
    key_bv = BitVector( textstring = key )
    return keysize,key_bv

main()
```

- Shown below is a terminal session with the code:

- AES Key size: 128

```
Enter key (any number of chars):  hello

Each 32-bit word of the key schedule is shown as a sequence of 4 one-byte integers:

word 0:  [104, 101, 108, 108]
word 1:  [111, 48, 48, 48]
word 2:  [48, 48, 48, 48]
word 3:  [48, 48, 48, 48]

word 4:  [109, 97, 104, 104]
word 5:  [2, 81, 88, 88]
word 6:  [50, 97, 104, 104]
word 7:  [2, 81, 88, 88]

word 8:  [190, 11, 2, 31]
word 9:  [188, 90, 90, 71]
word 10:  [142, 59, 50, 47]
word 11:  [140, 106, 106, 119]

word 12:  [184, 9, 247, 123]
word 13:  [4, 83, 173, 60]
word 14:  [138, 104, 159, 19]
word 15:  [6, 2, 245, 100]

word 16:  [199, 239, 180, 20]
word 17:  [195, 188, 25, 40]
word 18:  [73, 212, 134, 59]
word 19:  [79, 214, 115, 95]

word 20:  [33, 96, 123, 144]
word 21:  [226, 220, 98, 184]
word 22:  [171, 8, 228, 131]
word 23:  [228, 222, 151, 220]

word 24:  [28, 232, 253, 249]
word 25:  [254, 52, 159, 65]
word 26:  [85, 60, 123, 194]
word 27:  [177, 226, 236, 30]

word 28:  [196, 38, 143, 49]
word 29:  [58, 18, 16, 112]
word 30:  [111, 46, 107, 178]
word 31:  [222, 204, 135, 172]

word 32:  [15, 49, 30, 44]
word 33:  [53, 35, 14, 92]
word 34:  [90, 13, 101, 238]
word 35:  [132, 193, 226, 66]

word 36:  [108, 169, 50, 115]
word 37:  [89, 138, 60, 47]
word 38:  [3, 135, 89, 193]
word 39:  [135, 70, 187, 131]
```

```
word 40:  [0, 67, 222, 100]
word 41:  [89, 201, 226, 75]
word 42:  [90, 78, 187, 138]
word 43:  [221, 8, 0, 9]


Round keys in hex (first key for input block):

68656c6c6f30303030303030303030303
6d6168680251585832616868025158 58
be0b021fbc5a5a478e3b322f8c6a6a77
b809f77b0453ad3c8a689f130602f564
c7efb414c3bc192849d4863b4fd6735f
21607b90e2dc62b8ab08e483e4de97dc
1ce8fdf9fe349f41553c7bc2b1e2ec1e
c4268f313a1210706f2e6bb2decc87ac
0f311e2c35230e5c5a0d65ee84c1e242
6ca93273598a3c2f038759c18746bb83
0043de6459c9e24b5a4ebb8add080009
```

- What you see in hex above are 11 round keys, each 128 bit long, for the 10 rounds that will be used when the user supplied encryption key is 128 bits long. As you know by this time, the first round key listed above is what you need to XOR the input block with before any encryption rounds.

- Shown below is a Perl implementation of the Python script shown earlier in this subsection. Note that the Perl script requires you have installed at least Version 1.26 of the `Algorithm::BitVector` module.

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

##  gen_key_schedule.pl
##  Avi Kak  (February 2, 2018)

##  This script is for demonstrating the AES algorithm for generating the
##  key schedule.

##  It will prompt you for the key size, which must be one of 128, 192, 256.
```

```
##  It will also prompt you for a key.  If the key you enter is shorter
##  than what is needed for the AES key size, we add zeros on the right of
##  the key so that its length is as needed by the AES key size.

use strict;
use warnings;
use Algorithm::BitVector 1.26;

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

my @key_words;
my ($keysize, $key_bv) = get_key_from_user();

if ($keysize == 128) {
    @key_words = gen_key_schedule_128($key_bv);
} elsif ($keysize == 192) {
    @key_words = gen_key_schedule_192($key_bv);
} elsif ($keysize == 256) {
    @key_words = gen_key_schedule_256($key_bv);
} else {
    die "wrong keysize --- aborting";
}

my @key_schedule;
print "\nEach 32-bit word of the key schedule is shown as a sequence of 4 one-byte integers:\n";

foreach my $word_index (0..@key_words-1) {
    my $word = $key_words[$word_index];
    my @keyword_in_ints;
    foreach my $i (0..3) {
        push @keyword_in_ints, int( $word->get_slice([$i*8..($i+1)*8]) )
    }
    if ($word_index % 4 == 0) {
        print "\n";
    }
    print "word $word_index: @keyword_in_ints\n";
    push @key_schedule, "@keyword_in_ints";
}

my $num_rounds;
if ($keysize == 128) { $num_rounds = 10; }
if ($keysize == 192) { $num_rounds = 12; }
if ($keysize == 256) { $num_rounds = 14; }

my @round_keys = (undef) x ($num_rounds+1);

foreach my $i (0..$num_rounds) {
   $round_keys[$i] = ($key_words[$i*4] + $key_words[$i*4+1] + $key_words[$i*4+2] +
                                                          $key_words[$i*4+3])->get_bitvector_in_hex();
}
print("\n\nRound keys in hex (first key for input block):\n\n");
foreach my $round_key (@round_keys) {
    print "$round_key\n";
}
```

```
##  This is the g() function you see in Figure 4 of Lecture 8.
sub gee {
    my ($keyword, $round_constant, $byte_sub_table) = @_;
    my $rotated_word = $keyword->deep_copy();
    $rotated_word = $rotated_word << 8;
    my $newword = Algorithm::BitVector->new(size => 0);
    foreach my $i (0..3) {
        $newword += Algorithm::BitVector->new(intVal =>
                        $byte_sub_table->[int($rotated_word->get_slice([8*$i..8*($i+1)]))], size => 8);
    }
    $newword->set_slice([0..8], $newword->get_slice([0..8]) ^ $round_constant);
    $round_constant = $round_constant->gf_multiply_modular(Algorithm::BitVector->new(intVal => 0x02),
                                                                                $AES_modulus, 8);
    return $newword, $round_constant;
}

sub gen_key_schedule_128 {
    my $key_bv = shift;
    my $byte_sub_table = gen_subbytes_table();
    #  We need 44 keywords in the key schedule for 128 bit AES.  Each keyword is 32-bits
    #  wide. The 128-bit AES uses the first four keywords to xor the input block with.
    #  Subsequently, each of the 10 rounds uses 4 keywords from the key schedule. We will
    #  store all 44 keywords in the list key_words in this function.
    my @key_words = (undef) x 44;
    my $round_constant = Algorithm::BitVector->new(intVal => 0x01, size => 8);
    ($key_words[0],$key_words[1],$key_words[2],$key_words[3]) =
                                                    map $key_bv->get_slice([$_*32..($_+1)*32]), 0..3;
    foreach my $i (4..43) {
        if ($i%4 == 0) {
            my $kwd;
            ($kwd, $round_constant) = gee($key_words[$i-1], $round_constant, $byte_sub_table);
            $key_words[$i] = $key_words[$i-4] ^ $kwd;
        } else {
            $key_words[$i] = $key_words[$i-4] ^ $key_words[$i-1];
        }
    }
    return @key_words;
}

sub gen_key_schedule_192 {
    my $key_bv = shift;
    my $byte_sub_table = gen_subbytes_table();
    #  We need 52 keywords (each keyword consists of 32 bits) in the key schedule for
    #  192 bit AES.  The 192-bit AES uses the first four keywords to xor the input
    #  block with.  Subsequently, each of the 12 rounds uses 4 keywords from the key
    #  schedule. We will store all 52 keywords in the following list:
    my @key_words = (undef) x 52;
    my $round_constant = Algorithm::BitVector->new(intVal => 0x01, size => 8);
    foreach my $i (0..5) {
        $key_words[$i] = $key_bv->get_slice([$i*32 .. ($i+1)*32]);
    }
    foreach my $i (6..51) {
        if ($i%6 == 0) {
            my $kwd;
```

```
            ($kwd, $round_constant) = gee($key_words[$i-1], $round_constant, $byte_sub_table);
            $key_words[$i] = $key_words[$i-6] ^ $kwd;
        } else {
            $key_words[$i] = $key_words[$i-6] ^ $key_words[$i-1];
        }
    }
    return @key_words;
}

sub gen_key_schedule_256 {
    my $key_bv = shift;
    my $byte_sub_table = gen_subbytes_table();
    #  We need 60 keywords (each keyword consists of 32 bits) in the key schedule for
    #  256 bit AES. The 256-bit AES uses the first four keywords to xor the input
    #  block with.  Subsequently, each of the 14 rounds uses 4 keywords from the key
    #  schedule. We will store all 60 keywords in the following list:
    my @key_words = (undef) x 60;
    my $round_constant = Algorithm::BitVector->new(intVal => 0x01, size => 8);
    foreach my $i (0..7) {
        $key_words[$i] = $key_bv->get_slice([$i*32 .. ($i+1)*32]);
    }
    foreach my $i (8..59) {
        if ($i%8 == 0) {
            my $kwd;
            ($kwd, $round_constant) = gee($key_words[$i-1], $round_constant, $byte_sub_table);
            $key_words[$i] = $key_words[$i-8] ^ $kwd;
        } elsif (($i - int($i/8)*8) < 4) {
            $key_words[$i] = $key_words[$i-8] ^ $key_words[$i-1];
        } elsif (($i - int($i/8)*8) == 4) {
            $key_words[$i] = Algorithm::BitVector->new( size => 0);
            foreach my $j (0..3) {
                $key_words[$i] += Algorithm::BitVector->new(intVal =>
                 int($byte_sub_table->[int($key_words[$i-1]->get_slice([8*$j..8*($j+1)]))]), size => 8);
            }
            $key_words[$i] = $key_words[$i] ^ $key_words[$i-8];
        } elsif ( (($i - int($i/8)*8) > 4) && (($i - int($i/8)*8) < 8) ) {
            $key_words[$i] = $key_words[$i-8] ^ $key_words[$i-1];
        } else {
            die "error in key scheduling algo for i = $i\n";
        }
    }
    return @key_words;
}

sub gen_subbytes_table {
    my @subBytesTable;                                      # SBox for encryption
    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);
    }
    return \@subBytesTable;
}

sub get_key_from_user {
    my ($key, $keysize);
    print "\nAES key size: ";
    while ( $keysize = <STDIN> ) {
        chomp $keysize;
        if (($keysize != 128) && ($keysize != 192) && ($keysize != 256)) {
            die "\nkeysize is wrong (must be one of 128, 192, or 256) --- aborting";
        }
        last;
    }
    print "\nEnter key (any number of chars): ";
    while ( $key = <STDIN> ) {
        chomp $key;
        last;
    }
    if (length $key < int($keysize/8)) {
        $key .= '0' x ($keysize/8 - length $key);
    }
    my $key_bv = Algorithm::BitVector->new( textstring => $key );
    return $keysize, $key_bv;
}
```

- Running the script shown above should yield exactly the same results as the Python script shown earlier in this subsection.

Back to TOC

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 firs 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 by 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 Differential 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 $\Delta X = X_1 \otimes X_2$ between them, into the cipher while observing the differences $\Delta Y = Y_1 \otimes Y_2$ between the corresponding ciphertext blocks. We refer to ΔX as

the input differential and ΔY 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`, 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.

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

##  find_differentials_correlations.pl

##  Avi Kak  (March 4, 2015)

##  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).

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 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);                        #(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 .. $M-1) {                                                          #(B2)
        print "\ni=$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)
    }                                                                                    #(E7)
    return $arr_ref;                                                                     #(E8)
}

###################### 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) {                                      #(F4)
        print "\n";                                                             #(F5)
        foreach my $j ($lower .. $upper - 1) {                                  #(F6)
            print "$differential_hist->[$i][$j] ";                              #(F7)
        }
    }
}

#  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";
    close OUTFILE;
my $argstring = <<"END";                                                        #(G28)
set xrange [$x_min:$x_max]
set yrange [$y_min:$y_max]
set view 80,15
set hidden3d
splot "$temp_file" with lines
```

```
END
    unless (defined $pause_time) {                                                    #(G29)
        my $hardcopy_name =  "output_histogram.png";                                  #(G30)
        my $plot1 = Graphics::GnuplotIF->new();                                       #(G31)
        $plot1->gnuplot_cmd( 'set terminal png', "set output \"$hardcopy_name\"");    #(G32)
        $plot1->gnuplot_cmd( $argstring );                                            #(G33)
        my $plot2 = Graphics::GnuplotIF->new(persist => 1);                           #(G34)
       $plot2->gnuplot_cmd( $argstring );                                             #(G35)
    } else {                                                                          #(G36)
        my $plot = Graphics::GnuplotIF->new();                                        #(G37)
        $plot->gnuplot_cmd( $argstring );                                             #(G38)
        $plot->gnuplot_pause( $pause_time );                                          #(G39)
    }
}
```

- For an accurate and complete calculation of the input/output differentials histogram, you'd need to change the value of `$M` in line (A2) to 256. That would result in a large 256×256 histogram of integer values. For the purpose of our explanation here, we will make do with `$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).

```
0 0 0 0 0 0 0 2 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 2 0 0 0 0 0
0 0 2 0 0 0 0 2 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 2 0 2 0 2 2 0 0 0
0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 2 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 0 0 0 0 0 2 0
2 2 0 0 0 2 0 0 0 0 0 0 0 0 0 2 0 0 0 2 0 0 0 0 0 0 0 0 2 0 0 2
2 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 2 0 2 0 2 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 2 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 2 0 0 0 0 0
0 2 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 2 0 0 0 0 0 0
0 0 2 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 2 0 2 0 0 0 0 0 0 0 0 0 0 0 2 0 0 2 0 0 0 0 0 0 0 2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 4 0 0 0 2 0 2 0 0 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0
0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 2
0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 2 0 0 2 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 2 2 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0
2 0 0 0 0 2 0 0 0 0 2 0 0 0 2 0 0 2 0 0 0 2 0 0 0 0 0 0 2 0 0 0
0 0 2 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 2 0 2 0 0 0 0 0 0 0 2 0 0
0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 2 0 0 0 2 0
0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 2 0 0 0 0 0 0 0 2 4 0 0 0 0 0 2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 2 0 0 0 0 0 0 0 0 0 0 0 2 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 2 0
0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 2 0 0 0 0 0 0 0 2 0 0
0 0 0 0 0 2 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 0 0 0 0 0 0 0 0
2 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0
0 2 0 0 2 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 2 0 2 0 0 0 0 0 0 0 0 0
0 2 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0
0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 0
0 0 0 2 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 2 0 0 2 0 0 0 0 0 0 0 0 0
```

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

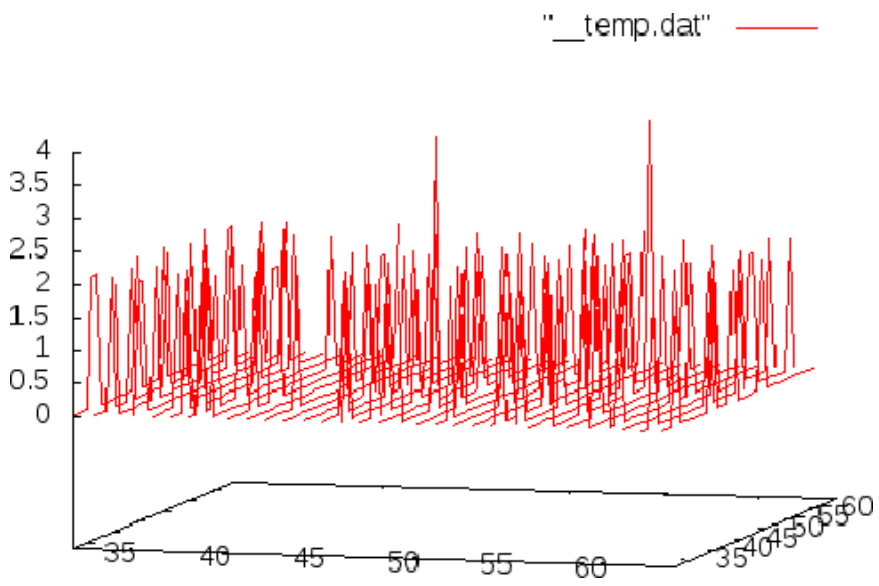

Figure 5: *Shown is a portion of the histogram of the input/output differentials for an SBox consisting of MI in* $GF(2^8)$. *(This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak)*

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 ΔX, you would want the output differential ΔY to be distributed uniformly

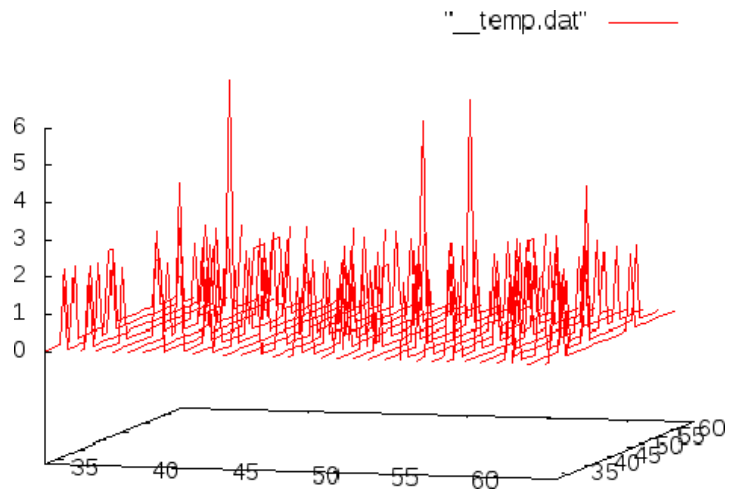

Figure 6: *Shown is a portion of the histogram of the input/output differentials for an SBox that carries out byte substitutions by looking up the table supplied in line (A9).* (*This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak*)

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 ΔX occurs in multiples of 2 since $\Delta X = X_i \otimes X_j = X_j \otimes X_i$. **(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 256×256 bins of the histogram is clearly not achievable — even theoretically.]

- As the value of the variable `$M` in line (A2) of the script `find_differen tials_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 somewhat larger.

- The Perl script that follows, `differential_attack_toy_example.pl`, is a demonstration of how the sort of non-uniformities in the histogram of the input/output differentials can be exploited to recover 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.

```
#!/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 substituion 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 @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);                      #(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 "\nworst input differentials: @worst_input_differentials\n";             #(A26)
    print "\nworst output differentials: @worst_output_differentials\n";           #(A27)
}

#  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) {                                                 #(A31)
    my @vote_sorted_keys =
            sort {$votes_for_keys{$b} <=> $votes_for_keys{$a}} keys %votes_for_keys;     #(A32)
    print "\nDisplaying the keys with the largest number of votes: @vote_sorted_keys[0..9]\n";
                                                                                   #(A33)
}

###################################  Subroutines  ##########################################

#  The differential attack:
```

```
sub estimate_last_round_key {                                                    #(B1)
    my $attempted = 0;                                                           #(B2)
    foreach my $i (2..255) {                                                     #(B3)
        print "+ " if $debug;                                                    #(B4)
        my $plaintext1 = Algorithm::BitVector->new(intVal => $i, size => 8);     #(B5)
        foreach my $j (2..255) {                                                 #(B6)
            my $plaintext2 = Algorithm::BitVector->new(intVal => $j, size => 8); #(B7)
            my $input_differential = $plaintext1 ^ $plaintext2;                  #(B8)
            next if int($input_differential) < 2;                                #(B9)
            next unless exists $worst_differentials{int($input_differential)};   #(B10)
            print "- " if $debug;                                                #(B11)
            my ($ciphertext1, $ciphertext2) =                                    #(B12)
              (encrypt($plaintext1, $encryption_key), encrypt($plaintext2, $encryption_key));
            my $output_differential = $ciphertext1 ^ $ciphertext2;               #(B13)
            next if int($output_differential) < 2;                               #(B14)
            last if $attempted++ > $tries;                                       #(B15)
            print " attempts made $attempted " if $attempted % 500 == 0;         #(B16)
            print "| " if $debug;                                                #(B17)
            foreach my $key (0..255) {                                           #(B18)
                print ". " if $debug;                                            #(B19)
                my $key_bv = Algorithm::BitVector->new(intVal => $key, size => 8);  #(B20)
                my $partial_decrypt_int1 = $decryption_inverses{int($ciphertext1 ^ $key_bv )};
                                                                                 #(B21)
                my $partial_decrypt_int2 = $decryption_inverses{int($ciphertext2 ^ $key_bv )};
                                                                                 #(B22)
                my $delta = $partial_decrypt_int1 ^ $partial_decrypt_int2;       #(B23)
                if (exists $worst_differentials{$delta}) {                       #(B24)
                    print "  voted  " if $debug;                                #(B25)
                    $votes_for_keys{$key}++;                                     #(B26)
                }
            }
        }
    }
}

sub encrypt {                                                                    #(C1)
    my $plaintext = shift;              # must be a bitvector                    #(C2)
    my $key = shift;                    # must be a bitvector                    #(C3)
    my $round_input = $plaintext;                                                #(C4)
    my $round_output;                                                            #(C5)
    my $round_key = $key;                                                        #(C6)
    if ($number_of_rounds > 1) {                                                 #(C7)
        foreach my $round (0..$number_of_rounds-1) {                             #(C8)
            $round_output = get_sbox_output_lookup($round_input) ^ $round_key;   #(C9)
            $round_input = $round_output;                                        #(C10)
            $round_key = $round_key >> 1;                                        #(C11)
        }
    } else {                                                                     #(C12)
        $round_output = get_sbox_output_lookup($round_input) ^ $key;             #(C13)
    }
    return $round_output;                                                        #(C14)
}

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

#  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.
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.
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),
                           Algorithm::BitVector->new(intVal => $j, size => 8));     #(G6)
            my $input_differential = int($a ^ $b);                                  #(G7)
            my ($c, $d) = (get_sbox_output_lookup($a), get_sbox_output_lookup($b)); #(B9)
            my $output_differential = int($c ^ $d);                                 #(G9)
            $differential_hist->[$input_differential][$output_differential]++;      #(G10)
        }
    }
}

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.
sub display_portion_of_histogram {                                                  #(J1)
    my $lower = shift;                                                              #(J2)
    my $upper = shift;                                                              #(J3)
    foreach my $i ($lower .. $upper - 1) {                                          #(J4)
        print "\n";                                                                 #(J5)
```

```
        foreach my $j ($lower .. $upper - 1) {                                  #(J6)
            print "$differential_hist->[$i][$j] ";                              #(J7)
        }
    }
}
```

- 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: [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.]

$$X_{i_1} \otimes X_{i_2} \ldots \otimes X_{i_m} \otimes Y_{i_1} \otimes Y_{i_2} \ldots \otimes Y_{i_n} \;=\; 0 \tag{5}$$

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

$$X_{i_1} \otimes X_{i_2} \ldots \otimes X_{i_m} \;=\; Y_{i_1} \otimes Y_{i_2} \ldots \otimes Y_{i_n} \tag{6}$$

- 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.

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

##  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.

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 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 {
    foreach my $x (0 .. 255) {               # specify a byte for the input to the SBox  #(B1)
        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)
              }
          }
      }
      foreach my $i (0 .. 255) {                                                       #(B24)
          foreach my $j (0 .. 255) {                                                   #(B25)
              $linear_approximation_table->[$i][$j] -= 128;                            #(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)
    }                                                                                  #(E7)
    return $arr_ref;                                                                   #(E8)
}

########################## Support Routines for Displaying LAT  ##############################
```

```
#  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.
sub display_portion_of_LAT {                                                    #(F1)
    my $lower = shift;                                                          #(F2)
    my $upper = shift;                                                          #(F3)
    foreach my $i ($lower .. $upper - 1) {                                      #(F4)
        print "\n";                                                             #(F5)
        foreach my $j ($lower .. $upper - 1) {                                  #(F6)
            print "$linear_approximation_table->[$i][$j] ";                     #(F7)
        }
    }
}

#  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.
sub plot_portion_of_LAT {                                                       #(G1)
    my $hist = shift;                                                           #(G2)
    my $lower = shift;                                                          #(G3)
    my $upper = shift;                                                          #(G4)
    my $pause_time = shift;                                                     #(G5)
    my @plot_points = ();                                                       #(G6)
    my $bin_width = my $bin_height = 1.0;                                       #(G7)
    my ($x_min, $y_min, $x_max, $y_max) = ($lower, $lower, $upper, $upper);     #(G8)
    foreach my $y ($y_min..$y_max-1) {                                          #(G9)
        foreach my $x ($x_min..$x_max-1) {                                      #(G10)
            push @plot_points, [$x, $y, $hist->[$y][$x]];                       #(G11)
        }
    }
    @plot_points = sort {$a->[0] <=> $b->[0]} @plot_points;                     #(G12)
    @plot_points = sort {$a->[1] <=> $b->[1] if $a->[0] == $b->[0]} @plot_points;  #(G13)
    my $temp_file = "__temp.dat";                                               #(G14)
    open(OUTFILE , ">$temp_file") or die "Cannot open temporary file: $!";      #(G15)
    my ($first, $oldfirst);                                                     #(G16)
    $oldfirst = $plot_points[0]->[0];                                           #(G17)
    foreach my $sample (@plot_points) {                                         #(G18)
        $first = $sample->[0];                                                  #(G19)
        if ($first == $oldfirst) {                                              #(G20)
            my @out_sample;                                                     #(G21)
            $out_sample[0] =  $sample->[0];                                     #(G22)
            $out_sample[1] =  $sample->[1];                                     #(G23)
            $out_sample[2] =  $sample->[2];                                     #(G24)
            print OUTFILE "@out_sample\n";                                      #(G25)
        } else {                                                                #(G26)
            print OUTFILE "\n";                                                 #(G27)
        }
        $oldfirst = $first;                                                     #(G28)
    }
```

```
    print OUTFILE "\n";
    close OUTFILE;
my $argstring = <<"END";                                                    #(G29)
set xrange [$x_min:$x_max]
set yrange [$y_min:$y_max]
set view 80,15
set hidden3d
splot "$temp_file" with lines
END
    unless (defined $pause_time) {                                          #(G30)
        my $hardcopy_name =  "LAT.png";                                     #(G31)
        my $plot1 = Graphics::GnuplotIF->new();                             #(G32)
        $plot1->gnuplot_cmd( 'set terminal png', "set output \"$hardcopy_name\"");  #(G33)
        $plot1->gnuplot_cmd( $argstring );                                  #(G34)
        my $plot2 = Graphics::GnuplotIF->new(persist => 1);                 #(G35)
       $plot2->gnuplot_cmd( $argstring );                                   #(G36)
    } else {                                                                #(G37)
        my $plot = Graphics::GnuplotIF->new();                              #(G38)
        $plot->gnuplot_cmd( $argstring );                                   #(G39)
        $plot->gnuplot_pause( $pause_time );                                #(G40)
    }
}
```

- 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 256×256. Theory dictates that the sum of the entries in each row or each column must be either 128 or -128.

```
128  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

```
 0   -6    8 -14    4    6   12    6   -2   12   -2   -4   -6   -8    2   -8 -12   -2   12   -2   -8 -14    0   -6   -2
 0    8   12    4   -8   -8   -4   12 -12 -12    8    8   -8    0   12 -12 -14   -6   -6   -6   -6   10   10    2    2
 0  -14    4    6   12   -2   -8    2   -2   12 -14    0   -2 -12   10    8   -2    8   -2    8    2    4   -6    4 -12
 0    4   -8   12 -12    8    0   12   -6   -6   10   10    2   -6   -2   -2    2   -6   -2   14   14    6   -2    6   -8
 0    6   -8   -2    8    6 -12 -14   -4   10 -12 -14   -8   -2    4   -6 -14    4    6   -8   -6   -4    2    4   -2
 0   12   -4   -8    0 -12    8 -12   -6   -6   -2   -2   -6   10   -6 -14   12 -12    8    0   12   12    4   -4   14
 0    6   12    2   12 -14 -12   10    8   -2    4   -6 -12   -6    4   10   12   -2    0   10   -8    2    8   -6   16
 0   -2 -12   -2   -6   -4   -6    8   -8   -6    4   -6   -2 -12   -2    8    8 -10    4   14   -6   12 -14   16 -12
 0   12 -12   12   -6   10   -6   -2   -6   -2    6   -2   -8    8    0   12    0    4    4   12 -14    2    2   -2   10
 0   -2    8 -14   10 -12   -2    4    4    6   -4    2   -2   12    2    4   -2   12    2    4    0   -6    0   -2    2
 0   -4    8    0   10 -14   -2   -6   -6   -2    2   -6   12    4    8   12    6 -14    2   10    0    0   -8   -4   -4
 0   -6   -8   -2    2   -8   -6 -12   -2   -8   -2   12   16   14   16   -6   -2 -12   10 -12    0    2   -4   10   12
 0   -8    0 -12   -6   -2   10   -6 -12    8   12    4   14   -2   -2    2   -6   14   -2   -2   -4 -12    0    4 -14
 0    2   12   10   -2    4   -6    4   -2    0    2    8   16   -2   12    6 -12   10    8   10   10    4   -2   -4   14
 0   -8 -12    8   -2   -6 -14   10    8   12    4   12   -6    2    6 -14   -8   12 -12   12   -2   -2   -6   -2    0
 0  -12 -14   -2    2 -14   12   12    8    0   -2    6   -2   -6 -12   -8   12   -8   10    6   -2   14   12 -12 -12
 0   -2   -6    8   -6    4 -12   -2 -10    4   12 -14 -12   14   10   12   -8   -2   -2    4   10   -4    8   10 -14
 0   12   -6   -2   -2    6    8    0    4    4    2    2   10   -2    8 -12   10   -2   -4   -8    0   -8   10    2    2
 0   -2   -6    8   14   -8    0   10   14   12    4   10 -12   -2   10   12    6    4   -8   -2   12 -10   14    0   12
 0   -8   -6    2   14   -6   12   -8   -6 -14    0    0    0   -4   10   -2   -2   10    0   12    4   -4    2   10   12
 0  -14   10    4    6   -4   12    2   12    2   -6    0    2 -12    4   -2   14   -4   -8 -10   -4   -2   10    4   -6
 0    0   10   -6   -2    2    4    8 -14    2    0   -8   -4    0   -2   -6   12    8   10   14    2   10    4 -12    6
 0   -6    2    4    6    4   -4   -6   16   -2   -2   -4   10    4   -4   -2 -12   10    2    0   10    4 -12   14    8
 0   -2    2 -12   -8   -2   14   16 -12   10    2   -4   12 -14   14    0 -12 -14    2   12   12   -6    6    8   -4
 0  -12   -6   -6 -12    8   -6   -6    6   14   12    0   -6 -14    4    8   12    8   -6    2    8    4   -6 -14   10
 0    6   -6 -12    4    2 -14   -4   -8   -2   -2   -8    8   -2   -6   -4 -10   -4    0   -6    2   -8   -8   10    6
 0  -12   -6   -6    8   12   -2   -2   14   -2 -12    0 -14    2    4    0    6   -6   -8   16    6   -6   -4    4    0
```

- 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 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

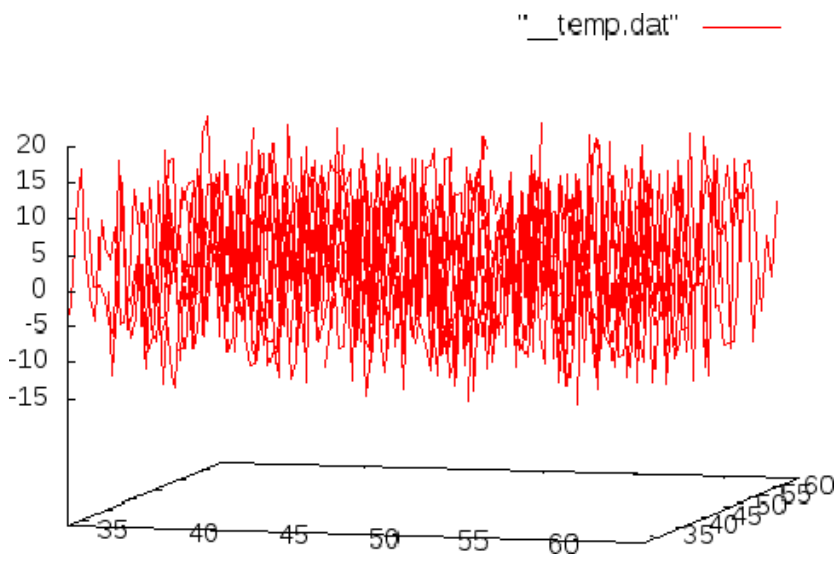

Figure 7: *Shown is a portion of the LAT for an SBox that calculates MIs in $GF(2^8)$ using the AES modulus.* (*This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak*)

compared to the SBox that is based on MI in $GF(2^8)$.

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

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

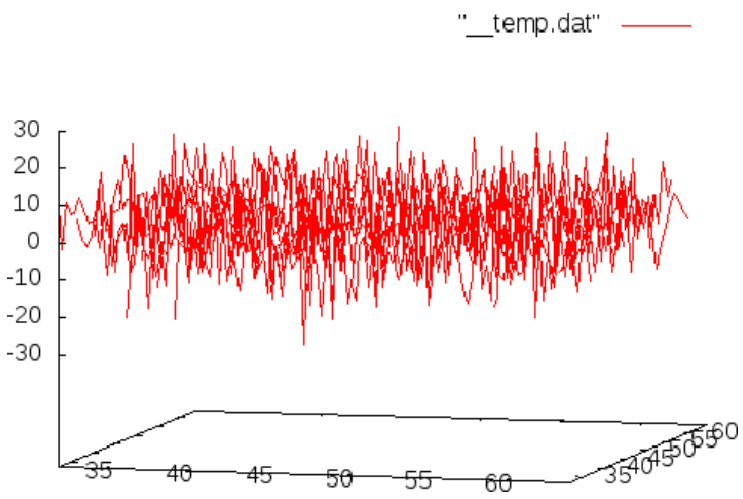

Figure 8: *Shown is a portion of the LAT for an SBox that carries out byte substitutions by looking up the table supplied in line (A8) of the LAT generator script.* (*This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak*)

- 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:

$$\begin{aligned} &P_{i_1} \otimes P_{i_2} \otimes \ldots \otimes Y_{1,j_1} \otimes \ldots Y_{1,j_1} \otimes \ldots Y_{1,j_m} \otimes \ldots K_1 \otimes Y_{2,j_1} \otimes \ldots Y_{2,j_1} \otimes \ldots \\ &\ldots K_2 \otimes Y_{3,j_1} \otimes \ldots Y_{3,j_1} \otimes \ldots\ldots Y_{n-1,j_1} \ldots Y_{n-1,j_m} \quad = \quad 0 \end{aligned}$$

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:

$$\begin{aligned} &XOR\ sum\ of\ only\ P\ and\ Y\ variables \quad \otimes \\ &\qquad XOR\ sum\ of\ key\ bits\ in\ rounds\ from\ 1\ through\ n-1 \quad = \quad 0 \end{aligned}$$

which can be abbreviated to

$$XOR\ sum\ of\ only\ P\ and\ Y\ variables \quad \otimes \quad \Sigma_K \quad = \quad 0$$

where Σ_K is the linear form that involves only the key bits from the first $n-1$ rounds.

- We have only two possibilities for Σ_K. Either it is equal to 0 or to 1. If we assume that both are equiprobable, that eliminates the influence of Σ_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.

Back to TOC

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 16×16 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 16×16 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

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

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:

`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:

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

The other `BitVector` method that should prove particularly useful for this homework is:

`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

`BitVector` class, when the modulus bit pattern is *mod*, you invoke

```
a.gf_multipy_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.