Lecture 3: Block Ciphers and the Data Encryption Standard

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

January 21, 2017
6:49pm

Goals:

- To introduce the notion of a block cipher in the modern context.
- To talk about the infeasibility of **ideal block ciphers**
- To introduce the notion of the **Feistel Cipher Structure**
- To go over **DES**, the Data Encryption Standard
- **To illustrate some of the DES steps with Python code**

CONTENTS

3.1: IDEAL BLOCK CIPHER

- In a modern block cipher (but still using a classical encryption method), we replace a block of N bits from the plaintext with a block of N bits from the ciphertext. This general idea is illustrated in Figure 1 for the case of $N = 4$. (In general, though, N is set to 64 or multiples thereof.)

- To understand Figure 1, note that there are 16 different possible 4-bit patterns. We can represent each pattern by an integer between 0 and 15. So the bit pattern 0000 could be represented by the integer 0, the bit pattern 0001 by integer 1, and so on. The bit pattern 1111 would be represented by the integer 15.

- In an ideal block cipher, the relationship between the input blocks and the output block is completely random. But it must be invertible for decryption to work. Therefore, it has to be one-to-one, meaning that each input block is mapped to a unique output block.

- The mapping from the input bit blocks to the output bit blocks can also be construed as a mapping from the *integers* correspond-

ing to the input bit blocks to the *integers* corresponding to the output bit blocks.

- The encryption key for the ideal block cipher is the codebook itself, meaning the table that shows the relationship between the input blocks and the output blocks.

- Figure 1 depicts an ideal block cipher that uses blocks of size 4. Each block of 4 bits in the plaintext is transformed into a block of 4 ciphertext bits.

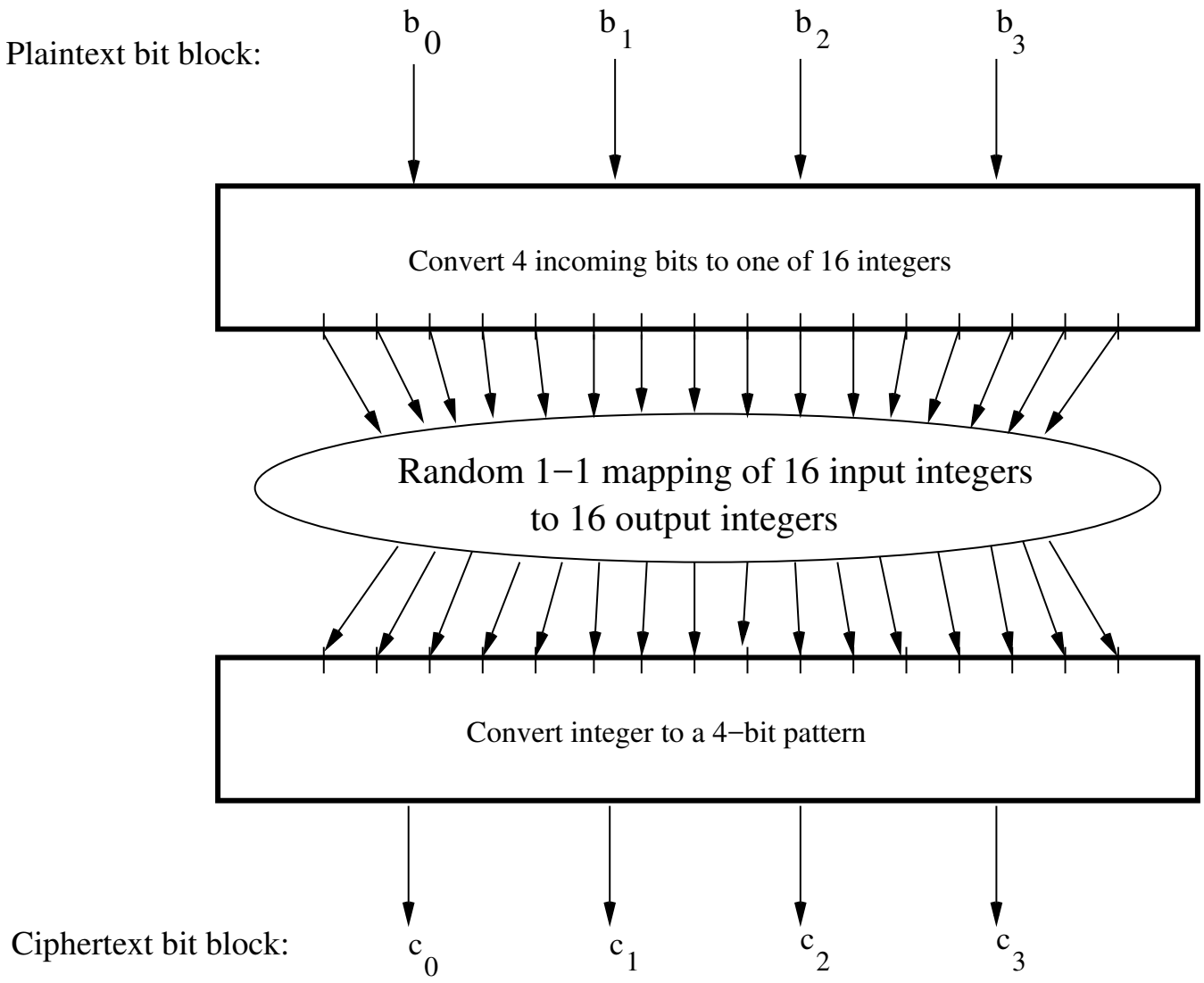

Figure 1: *The ideal block cipher when the block size equals 4 bits.* (*This figure is from Lecture 3 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

3.1.1: The Size of the Encryption Key for the Ideal Block Cipher

- Consider the case of 64-bit block encryption.

- With a 64-bit block, we can think of each possible input block as one of 2^{64} integers and for each such integer we can specify an output 64-bit block. We can construct the codebook by displaying just the output blocks in the order of the integers corresponding to the input blocks. Such a code book will be of size $64 \times 2^{64} \approx 10^{21}$.

- That implies that the encryption key for the ideal block cipher using 64-bit blocks will be of size 10^{21}.

- The size of the encryption key would make the ideal block cipher an impractical idea. Think of the logistical issues related to the transmission, storage, and processing of such large keys.

3.2: The Feistel Structure for Block Ciphers

The DES algorithm for encryption and decryption, which is the main theme of this lecture, is based on what is known as the Feistel Structure. This section and the next two subsections introduce this structure:

- Named after the IBM cryptographer Horst Feistel and first implemented in the Lucifer cipher by Horst Feistel and Don Coppersmith.

- A cryptographic system based on Feistel structure uses the same basic algorithm for both encryption and decryption.

- As shown in Figure 2, the Feistel structure consists of multiple rounds of processing of the plaintext, with each round consisting of a "substitution" step followed by a permutation step.

- The input block to each round is divided into two halves that we can denote **L** and **R** for the left half and the right half.

- In each round, the right half of the block, **R**, goes through unchanged. But the left half, **L**, goes through an operation that depends on **R** and the encryption key. The operation carried out on the left half **L** is referred to as the Feistel Function.

- The permutation step at the end of each round consists of swapping the modified **L** and **R**. *Therefore, the* **L** *for the next round would be* **R** *of the current round. And* **R** *for the next round be the output* **L** *of the current round.*

- The next two subsection present important properties of the Feistel structure. As you will see, these properties are invariant to our choice for the Feistel Function.

- Besides DES, there exist several block ciphers today — the most popular of these being Blowfish, CAST-128, and KASUMI — that are also based on the Feistel structure.

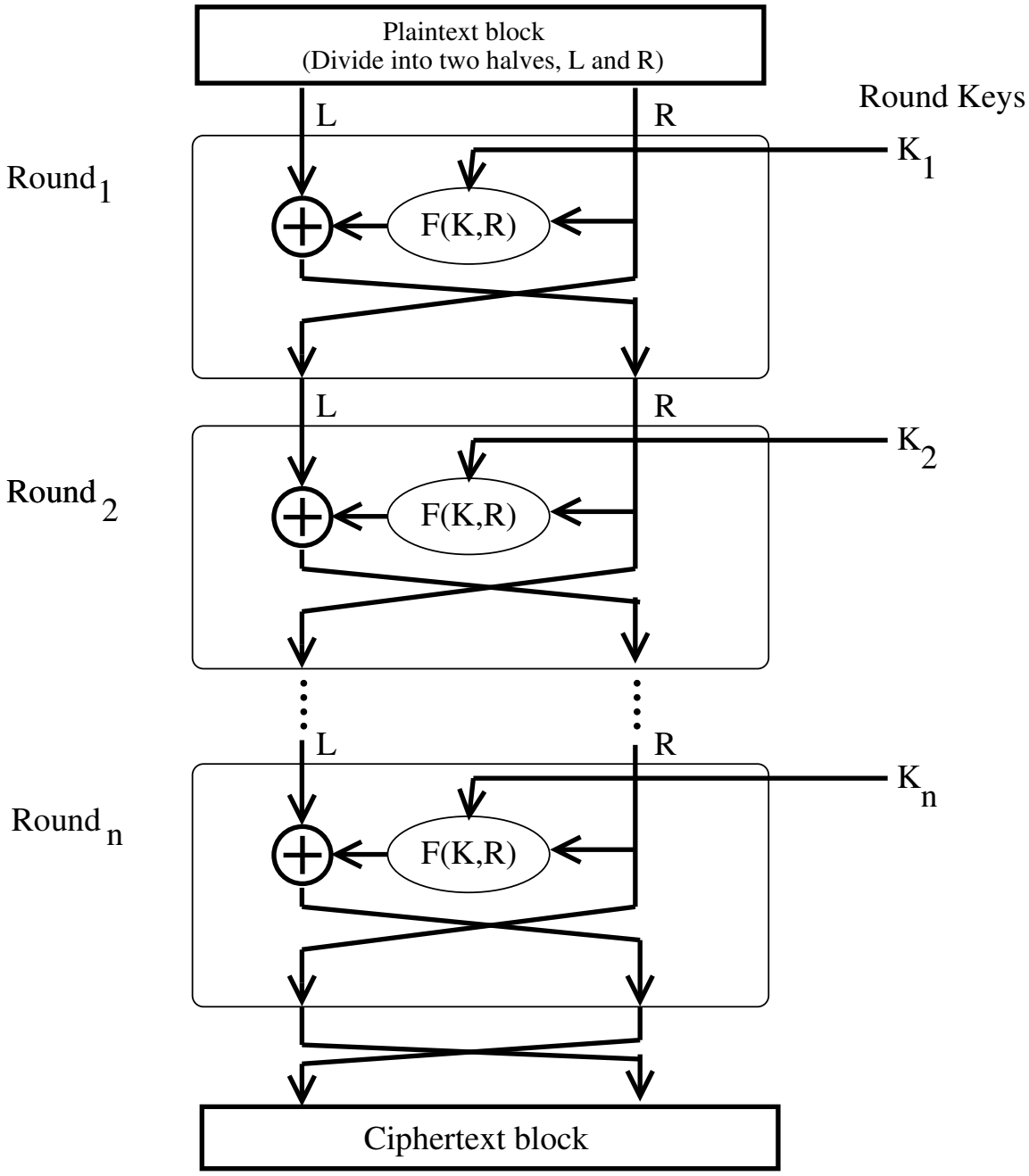

Figure 2: *The Feistel Structure for symmetric key cryptography* (*This figure is from Lecture 3 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

3.2.1: Mathematical Description of Each Round in the Feistel Structure

- Let LE_i and RE_i denote the output half-blocks at the end of the i^{th} round of processing. The letter 'E' denotes encryption.

- In the Feistel structure, the relationship between the output of the i^{th} round and the output of the previous round, that is, the $(i-1)^{th}$ round, is given by

$$\begin{aligned} LE_i &= RE_{i-1} \\ RE_i &= LE_{i-1} \oplus F(RE_{i-1}, K_i) \end{aligned}$$

  where $\oplus$ denotes the bitwise EXCLUSIVE OR operation. The symbol F denotes the operation that "scrambles" RE_{i-1} of the previous round with what is shown as the **round key** K_i in Figure 2. The round key K_i is derived from the main encryption key as we will explain later.

- F is referred to as the Feistel function, after Horst Feistel naturally.

- Assuming 16 rounds of processing (which is typical), the output of the last round of processing is given by

$$
\begin{aligned}
LE_{16} &= RE_{15} \\
RE_{16} &= LE_{15} \oplus F(RE_{15}, K_{16})
\end{aligned}
$$

3.2.2: Decryption in Ciphers Based on the Feistel Structure

- As shown in Figure 3, the decryption algorithm is exactly the same as the encryption algorithm with the only difference that the round keys are used in the reverse order.

- **The output of each round during decryption is the input to the corresponding round during encryption — except for the left-right switch between the two halves. This property holds true regardless of the choice of the Feistel function F.**

- To prove the above claim, let LD_i and RD_i denote the left half and the right half of the output of the i^{th} round.

- That means that the output of the first decryption round consists of LD_1 and RD_1. So we can denote the input to the first decryption round by LD_0 and RD_0. The relationship between the two halves that are input to the first decryption round and what is output by the encryption algorithm is:

$$\begin{aligned} LD_0 &= RE_{16} \\ RD_0 &= LE_{16} \end{aligned}$$

- We can write the following equations for the output of the first decryption round

$$\begin{aligned} LD_1 &= RD_0 \\ &= LE_{16} \\ &= RE_{15} \end{aligned}$$

$$\begin{aligned} RD_1 &= LD_0 \oplus F(RD_0, K_{16}) \\ &= RE_{16} \oplus F(LE_{16}, K_{16}) \\ &= [LE_{15} \oplus F(RE_{15}, K_{16})] \oplus F(RE_{15}, K_{16}) \\ &= LE_{15} \end{aligned}$$

  This shows that, except for the left-right switch, the output of the first round of decryption is the same as the input to the last stage of the encryption round since we have $LD_1 = RE_{15}$ and $RD_1 = LE_{15}$

- The following equalities are used in the above derivation. Assume that A, B, and C are bit arrays.

$$[A \oplus B] \oplus C = A \oplus [B \oplus C]$$

$$\begin{aligned} A \oplus A &= 0 \\ A \oplus 0 &= A \end{aligned}$$

- **The above result is independent of the precise nature of the Feistel function** F. That is, the output of each round during decryption is the input to the corresponding round during encryption for every choice of the Feistel function F.

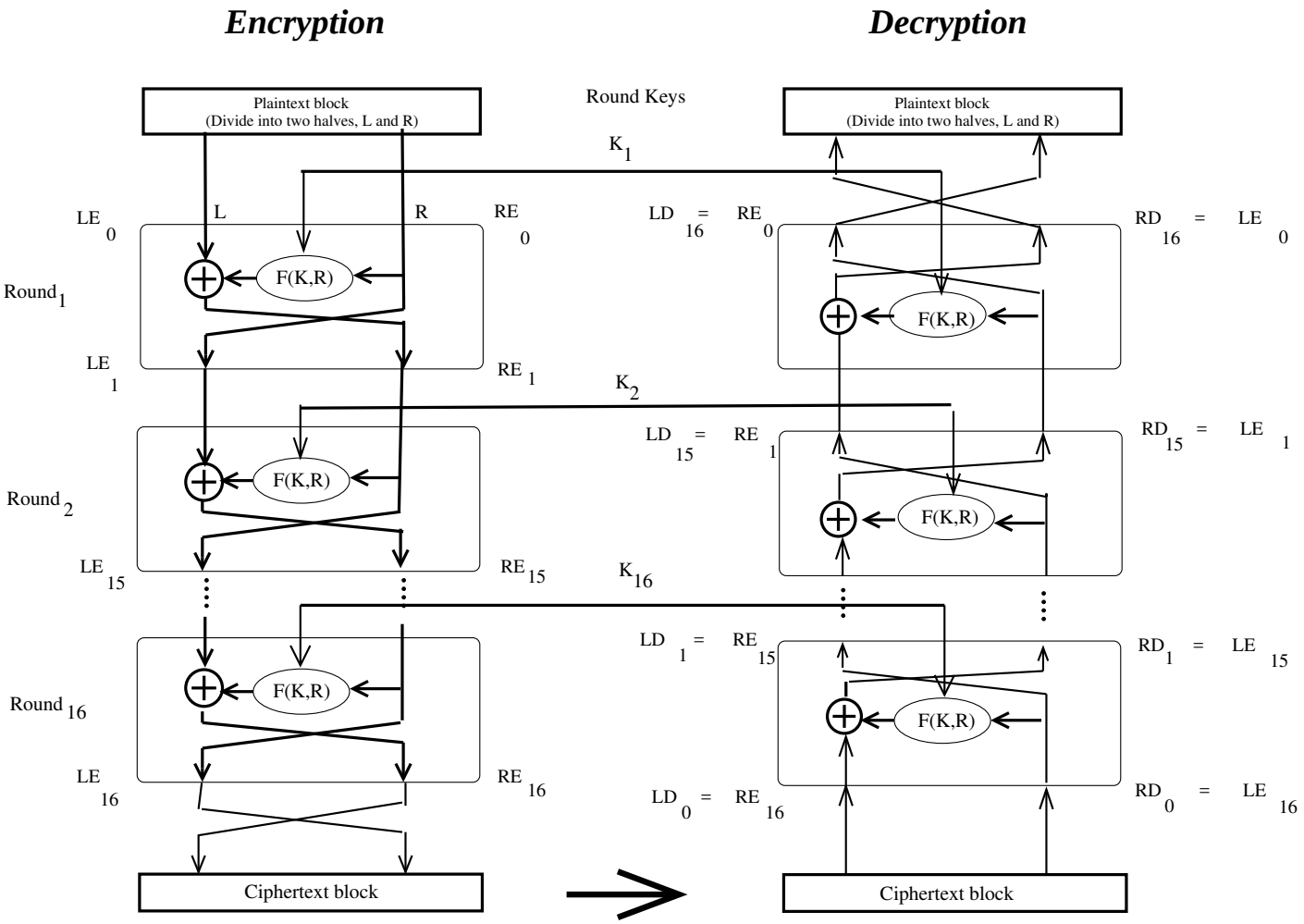

Figure 3: *When a Feistel structure is used, decryption works the same as encryption.* (*This figure is from Lecture 3 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

3.3: DES: THE DATA ENCRYPTION STANDARD

- Adopted by NIST in 1977.

- Based on a cipher (Lucifer) developed earlier by IBM for Lloyd's of London for cash transfer.

- DES uses the Feistel cipher structure with 16 rounds of processing.

- DES uses a 56-bit encryption key. (The key size was apparently dictated by the memory and processing constraints imposed by a single-chip implementation of the algorithm for DES.) The key itself is specified with 8 bytes, but one bit of each byte is used as a parity check.

- **DES encryption was broken in 1999 by Electronics Frontiers Foundation (EFF, `www.eff.org`).** This resulted in NIST issuing a new directive that year that required organizations to use **Triple DES**, that is, three consecutive applications

of **DES**. (That DES was found to be not as strong as originally believed also prompted NIST to initiate the development of new standards for data encryption. The result is **AES** that we will discuss later.)

- **Triple DES** continues to enjoy wide usage in commercial applications even today. To understand Triple DES, you must first understand the basic **DES** encryption.

- As mentioned, DES uses the Feistel structure with 16 rounds.

- What is specific to DES is the implementation of the F function in the algorithm and how the round keys are derived from the main encryption key.

- As will be explained in Section 3.3.5, the round keys are generated from the main key by a sequence of permutations. Each round key is 48 bits in length.

3.3.1: One Round of Processing in DEA

- The algorithmic implementation of DES is known as **DEA** for **Data Encryption Algorithm.**

- Figure 4 shows a single round of processing in DEA. The dotted rectangle constitutes the F function.

- The 32-bit right half of the 64-bit input data block is expanded by into a 48-bit block. This is referred to as the **expansion permutation** step, or the **E-step**.

- The above-mentioned E-step entails the following:
  - first divide the 32-bit block into eight 4-bit words
  - attach an additional bit on the left to each 4-bit word that is the last bit of the previous 4-bit word
  - attach an additional bit to the right of each 4-bit word that is the beginning bit of the next 4-bit word.

  Note that what gets prefixed to the first 4-bit block is the last bit of the last 4-bit block. By the same token, what gets appended to the last 4-bit block is the first bit of the first 4-bit block. The

reason for why we expand each 4-bit block into a 6-bit block in the manner explained will become clear shortly.

- The 56-bit key is divided into two halves, each half shifted separately, and the combined 56-bit key **permuted/contracted** to yield a 48-bit **round** key. How this is done will be explained later.

- The 48 bits of the expanded output produced by the E-step are XORed with the round key. This is referred to as **key mixing**.

- The output produced by the previous step is broken into eight six-bit words. Each six-bit word goes through a substitution step; its replacement is a 4-bit word. The substitution is carried out with an **S-box**, as explained in greater detail in Section 3.3.2. [The name "S-Box" stands for "Substitution Box".]

- So after all the substitutions, we again end up with a 32-bit word.

- The 32-bits of the previous step then go through a P-box based permutation, as shown in Figure 4.

- What comes out of the P-box is then XORed with the left half of the 64-bit block that we started out with. The output of this

XORing operation gives us the right half block for the next round.

- Note that the goal of the substitution step implemented by the **S-box** is to introduce **diffusion** in the generation of the output from the input. *Diffusion means that each plaintext bit must affect as many ciphertext bits as possible.*

- The strategy used for creating the different round keys from the main key is meant to introduce **confusion** into the encryption process. *Confusion in this context means that the relationship between the encryption key and the ciphertext must be as complex as possible.* Another way of describing confusion would be that each bit of the key must affect as many bits as possible of the output ciphertext block.

- Diffusion and confusion are the two cornerstones of block cipher design.

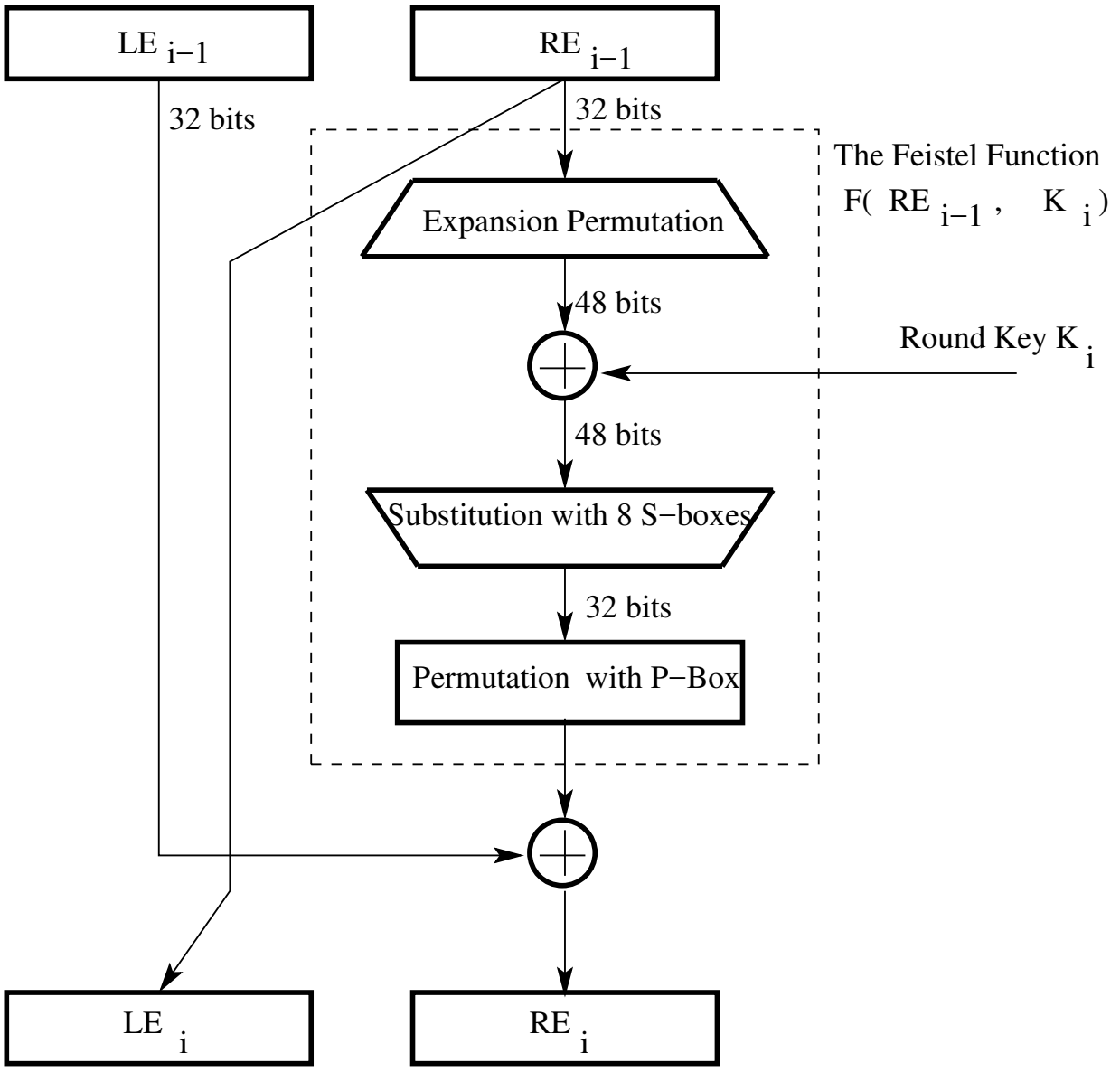

Figure 4: *One round of processing in DES.* (*This figure is from Lecture 3 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

3.3.2: The S-Box for the Substitution Step in Each Round

- As shown in Figure 5, the 48-bit input word is divided into eight 6-bit words and each 6-bit word fed into a separate S-box. Each S-box produces a 4-bit output. Therefore, the 8 S-boxes together generate a 32-bit output. As you can see, the overall substitution step takes the 48-bit input back to a 32-bit output.

- Each of the eight S-boxes consists of a 4×16 table lookup for an output 4-bit word. The first and the last bit of the 6-bit input word are decoded into one of 4 rows and the middle 4 bits decoded into one of 16 columns for the table lookup.

- The goal of the substitution carried out by an S-box is to enhance **diffusion**, as mentioned previously. As you will recall from the E-step described in Section 3.3.1, the expansion-permutation step (the E-step) expands a 32-bit block into a 48-bit block by attaching a bit at the beginning and a bit at the end of each 4-bit sub-block, the two bits needed for these attachments belonging to the adjacent blocks.

- Thus, the row lookup for each of the eight S-boxes becomes a function of the input bits for the previous S-box and the next

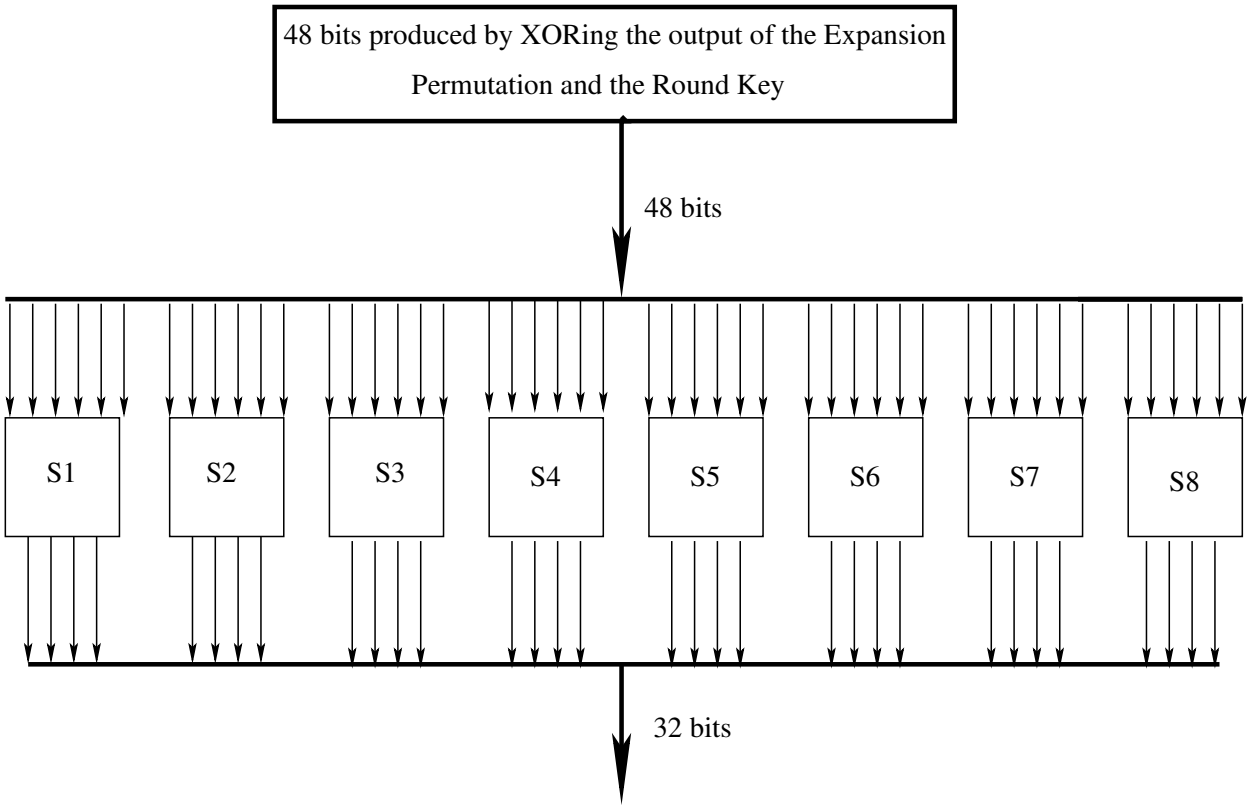

Figure 5: *The 48 bits coming out of the expansion permutation are first XORed with the round key and then, as shown, fed into the 8 S-boxes of DES.* (*This figure is from Lecture 3 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

S-box.

- In the design of the DES, the S-boxes were tuned to enhance the resistance of DES to what is known as the **differential cryptanalysis** attack, or, sometimes more briefly as **differential attack**. [As will be explained in much greater detail (and also demonstrated) in Section 8.9 of Lecture 8, differential cryptanalysis of block ciphers consists of presenting to the encryption algorithm pairs of plaintext bit patterns with known differences between them and examining the differences between the corresponding cyphertext outputs. The outputs are usually recorded at the input to the last round of the cipher. Let's represent one plaintext bit block by $X = [X_1, X_2, ..., X_n]$ where X_i denotes the i^{th} bit in the block, and let's represent the corresponding output bit block by $Y = [Y_1, Y_2, ..., Y_n]$. By the difference between two plaintext bit blocks X' and X'' we mean $\Delta X = X' \oplus X''$. The difference between the corresponding outputs Y' and Y'' is given by $\Delta Y = Y' \oplus Y''$. The pair $(\Delta X, \Delta Y)$ is known as a **differential**. In an ideally randomizing block cipher, the probability of ΔY being a particular value for a given ΔX is $1/2^n$ for an n-bit block cipher. What is interesting is that the probabilities of ΔY taking on different values for a given ΔX can be shown to be independent of the encryption key because of the properties of the XOR operator, but these probabilities are strongly dependent on the S-box tables. By feeding into a cipher several pairs of plaintext blocks with known ΔX and observing the corresponding ΔY, it is possible to establish constraints on the round key bits encountered along the different paths in the encryption processing chain. (By constraints I mean the following: Speaking hypothetically for the purpose of illustrating a point and focusing on just one round of DES, suppose we can show that the following condition can be expected to be obeyed with high probability: $\Delta X_i \oplus \Delta Y_i \oplus K_i = 0$ for some bit K_i of the encryption key, then it must be the case that $K_i = \Delta X \oplus \Delta Y$.) Note that differential cryptanalysis is a **chosen plaintext attack**, meaning that the attacker will feed known plaintext bit patterns into the cipher and analyze the corresponding outputs in order to figure out the encryption key. In a theoretical analysis of an attack based on differential cryptanalysis, it was shown by Eli Biham and Adi Shamir in 1990 that the DES's encryption key could be figured out provided one

could feed known 2^{47} plaintext blocks into the cipher. For a tutorial by Howard Heys on differential cryptanalysis, see `http://www.engr.mun.ca/~howard/PAPERS/ldc_tutorial.pdf`. The title of the tutorial is "A Tutorial on Linear and Differential Cryptanalysis."]

3.3.3: The Substitution Tables

- Shown on the next page are the eight S-boxes, S_1 through S_8, each S-box being a 4×16 substitution table that is used to convert 6 incoming bits into 4 outgoing bits.

- As mentioned earlier, each row of a substitution table is indexed by the two outermost bits of a six-bit block and each column by the remaining inner 4 bit.

The 4×16 substitution table for S-box S_1															
14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S-box S_2															
15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
S-box S_3															
10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
S-box S_4															
7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
S-box S_5															
2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
S-box S_6															
12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
S-box S_7															
4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
S-box S_8															
13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

- The Python code shown below illustrates how you can use the eight S-boxes for the substitutions you need for the right half of the input in each round:

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

##  illustrate_des_substitution.py

##  Avi Kak
##  January 21, 2017

##  This is a demonstration of how you can carry out S-boxes based substitution
##  in DES.  The code shown implements the "Substitution with 8 S-boxes" step
##  that you see inside the dotted Feistel function in Figure 4 of Lecture 3 notes.

##  IMPORTANT:  This demonstration code does NOT include XORing with the round
##              key that must be carried out on the expanded right-half block
##              before it is subject to the S-boxes based substitution step
##              shown here.

from BitVector import *

expansion_permutation = [31,  0,  1,  2,  3,  4,
                          3,  4,  5,  6,  7,  8,
                          7,  8,  9, 10, 11, 12,
                         11, 12, 13, 14, 15, 16,
                         15, 16, 17, 18, 19, 20,
                         19, 20, 21, 22, 23, 24,
                         23, 24, 25, 26, 27, 28,
                         27, 28, 29, 30, 31, 0]

s_boxes = {i:None for i in range(8)}

s_boxes[0] = [ [14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7],
               [0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8],
               [4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0],
               [15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13] ]

s_boxes[1] = [ [15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10],
               [3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5],
               [0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15],
               [13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9] ]

s_boxes[2] = [ [10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8],
               [13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1],
               [13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7],
               [1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12] ]

s_boxes[3] = [ [7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15],
               [13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9],
               [10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4],
```

```
                [3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14] ]

s_boxes[4] = [ [2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9],
                [14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6],
                [4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14],
                [11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3] ]

s_boxes[5] = [ [12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11],
                [10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8],
                [9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6],
                [4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13] ]

s_boxes[6] = [ [4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1],
                [13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6],
                [1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2],
                [6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12] ]

s_boxes[7] = [ [13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7],
                [1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2],
                [7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8],
                [2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11] ]

def substitute( expanded_half_block ):
    '''
    This method implements the step "Substitution with 8 S-boxes" step you see inside
    Feistel Function dotted box in Figure 4 of Lecture 3 notes.
    '''
    output = BitVector (size = 32)
    segments = [expanded_half_block[x*6:x*6+6] for x in range(8)]
    for sindex in range(len(segments)):
        row = 2*segments[sindex][0] + segments[sindex][-1]
        column = int(segments[sindex][1:-1])
        output[sindex*4:sindex*4+4] = BitVector(intVal = s_boxes[sindex][row][column], size = 4)
    return output

# For the purpose of this illustration, let's just make up the right-half of a
# 64-bit DES bit block:
right_half_32bits = BitVector( intVal = 800000700, size = 32 )

# Now we need to expand the 32-bit block into 48 bits:
right_half_with_expansion_permutation = right_half_32bits.permute( expansion_permutation )

print "expanded right_half_32bits: ", right_half_with_expansion_permutation

# The following statement takes the 48 bits back down to 32 bits after carrying
# out S-box based substitutions:
output = substitute(right_half_with_expansion_permutation)
print output
```

3.3.4: The P-Box Permutation in the Feistel Function

The last step in the Feistel function shown in Figure 4 is labeled "Permutation with P-Box". The permutation sequence is shown below. [It looks like a table, but it is not — as explained below]

P-Box Permutation							
15	6	19	20	28	11	27	16
0	14	22	25	4	17	30	9
1	7	23	13	31	26	2	8
18	12	29	5	21	10	3	24

- This permutation 'table' says that the 0^{th} output bit will be the 15^{th} bit of the input, the 1^{st} output bit the 6^{th} bit of the input, and so on, for all of the 32 bits of the output that are obtained from the 32 bits of the input.

- Do NOT associate any meaning with the row-organization of the table — except for the following: Each row of the table tells us how to select the input bits for the output byte corresponding to the row. For example, for the second output byte, the first entry in the second row means that the 0^{th} bit of the second output byte — meaning the 8^{th} bit of the output — will be the 0^{th} bit

of the 32-bit input. *Note that bit indexing is 0-based — as it would be in your Perl or Python script*

- Keep in mind the fact that, when using the `BitVector` module in Python or the `Algorithm::BitVector` module in Perl, a permutation such as the one shown above can be carried out with a one-line command. For example, in Python, the code fragment would look like:

```
sboxes_output =  BitVector representation of the
                          output of the S-Boxes
right_half = sboxes_output.permute( pbox_permutation )
```

  where **`permute()`** is a method defined for the **`BitVector`** class. The argument **`pbox_permutation`** you see above is the sequence of all the entries in the 'table' on the previous page expressed as a one-dimensional array.

3.3.5: The DES Key Schedule: Generating the Round Keys

- The initial 56-bit key may be represented as 8 bytes, with the last bit (the least significant bit) of each byte used as a parity bit.

- The relevant 56 bits are subject to a permutation at the beginning before any round keys are generated. This is referred to as Permutation Choice 1 that is shown in Section 3.3.6.

- At the beginning of each round, we divide the 56 relevant key bits into two 28 bit halves and circularly shift to the left each half by one or two bits, depending on the round, as shown in the table on the next page.

- For generating the round key, we join together the two halves and apply a 56 bit to 48 bit contracting permutation (this is referred to as Permutation Choice 2, as shown in Section 3.3.7) to the joined bit pattern. The resulting 48 bits constitute our round key.

- The contraction permutation shown in Permutation Choice 2, along with the one-bit or two-bit rotation of the two key halves

prior to each round, is meant to ensure that each bit of the original encryption key is used in roughly 14 of the 16 rounds.

- The two halves of the encryption key generated in each round are fed as the two halves going into the next round.

- The table shown below tells us how many positions to use for the left circular shift that is applied to the two key halves at the beginning of each round:

Round Number	*Number of left shifts*
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

- When using the `BitVector` module for programming in Python, or the `Algorithm::BitVector` module for programming in Perl, the steps described above for splitting the 56-bit key, circular-shifting each half separately, and then rejoining the two halves

can be carried simply by a command sequence that in Python looks like

```
[left,right] = key_bv.divide_into_two()
left   <<  shifts[i]
right  <<  shifts[i]
rejoined_key_bv = left + right
```

where **key_bv** is the **BitVector** representation of the 56-bit key entering the round and **shifts** is the array that consists of the second column entries in the table shown on the previous page. The method **divide_into_two()** is defined for the **BitVector** class.

- The Python code shown in Section 3.3.7 is an illustration of how you can implement the steps described above.

3.3.6: Initial Permutation of the Encryption Key

Permutation Choice 1						
56	48	40	32	24	16	8
0	57	49	41	33	25	17
9	1	58	50	42	34	26
18	10	2	59	51	43	35
62	54	46	38	30	22	14
6	61	53	45	37	29	21
13	5	60	52	44	36	28
20	12	4	27	19	11	3

- The bit indexing is based on using the range 0-63 for addressing the bit positions in an 8-byte bit pattern in which the last bit of each byte is used as a parity bit. [Note that each row shown above has has only 7 positions — the positions corresponding to the parity bit are NOT included above. That is, you will NOT see the positions 7, 15, etc., listed in the permutations shown. Nevertheless, the bit addressing spans the full 0-63 range.] The permutations shown above do not constitute a table, in the sense that the rows and the columns do NOT carry any special and separate meanings. The permutation order for the bits is given by reading the entries shown from the upper left corner to the lower right corner.

- This permutation tells us that the 0^{th} bit of the output will be

the 56^{th} bit of the input (in a 64 bit representation of the 56-bit encryption key), the 1^{st} bit of the output the 48^{th} bit of the input, and so on, until finally we have for the 55^{th} bit of the output the 3^{rd} bit of the input.

- When programming in Python using the `BitVector` module, or in Perl using the `Algorithm::BitVector` module, the permutations shown on the previous page can be carried out trivially by calling the `permute()` method of the modules. Using Python to illustrate, you could call

```
user_key_bv = BitVector( textstring = user-supplied_key )
key_bv = user_key_bv.permute( initial_permutation )
```

  where, as mentioned earlier, `permute()` is a method defined for the `BitVector` class and `initial_permutation` is the permutation shown on the previous slide expressed as a 1-dimensional array of integers.

- The code snippet shown below illustrates how you can create the 56-bit key from the eight characters supplied by the user.

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

## get_encryption_key.py

import sys
from BitVector import *

key_permutation_1 = [56,48,40,32,24,16,8,0,57,49,41,33,25,17,
                      9,1,58,50,42,34,26,18,10,2,59,51,43,35,
                     62,54,46,38,30,22,14,6,61,53,45,37,29,21,
```

```
                    13,5,60,52,44,36,28,20,12,4,27,19,11,3]

def get_encryption_key():
    key = ""
    while True:
        if sys.version_info[0] == 3:
            key = input("Enter a string of 8 characters for the key: ")
        else:
            key = raw_input("Enter a string of 8 characters for the key: ")
        if len(key) != 8:
            print("\nKey generation needs 8 characters exactly.  Try again.\n")
            continue
        else:
            break
    key = BitVector(textstring = key)
    key = key.permute(key_permutation_1)
    return key

key = get_encryption_key()
print("Here is the 56-bit encryption key generated from your input:\n")
print(key)
```

3.3.7: Contraction-Permutation that Generates the 48-Bit Round Key from the 56-Bit Key

Permutation Choice 2							
13	16	10	23	0	4	2	27
14	5	20	9	22	18	11	3
25	7	15	6	26	19	12	1
40	51	30	36	46	54	29	39
50	44	32	47	43	48	38	55
33	52	45	41	49	35	28	31

- As on the previous page, bit addressing shown above uses the full 0-63 range in an 8-byte pattern. Since the last bit of each byte is used as a parity bit, you will not see the bit positions 7, 15, 23, etc., in the permutation shown above.

- As with permutation shown on the previous page, what is shown above is NOT a table, in the sense that the rows and the columns do not carry any special and separate meanings. The permutation order for the bits is given by reading the entries shown from the upper left corner to the lower right corner.

- Since there are only six rows and there are 8 positions in each

row, the output will consist of 48 bits.

- When programming in Python using the `BitVector` class, the permutations shown on the previous page can be carried out trivially by calling the `permute()` method of the class, as mentioned earlier.

- The Python code shown below illustrates how you can generate all 16 round keys using the BitVector module:

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

## generate_round_keys.py

import sys
from BitVector import *

key_permutation_1 = [56,48,40,32,24,16,8,0,57,49,41,33,25,17,
                      9,1,58,50,42,34,26,18,10,2,59,51,43,35,
                     62,54,46,38,30,22,14,6,61,53,45,37,29,21,
                     13,5,60,52,44,36,28,20,12,4,27,19,11,3]

key_permutation_2 = [13,16,10,23,0,4,2,27,14,5,20,9,22,18,11,
                      3,25,7,15,6,26,19,12,1,40,51,30,36,46,
                     54,29,39,50,44,32,47,43,48,38,55,33,52,
                     45,41,49,35,28,31]

shifts_for_round_key_gen = [1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1]

def generate_round_keys(encryption_key):
    round_keys = []
    key = encryption_key.deep_copy()
    for round_count in range(16):
        [LKey, RKey] = key.divide_into_two()
        shift = shifts_for_round_key_gen[round_count]
        LKey << shift
        RKey << shift
        key = LKey + RKey
        round_key = key.permute(key_permutation_2)
        round_keys.append(round_key)
    return round_keys
```

```
def get_encryption_key():
    key = ""
    while True:
        if sys.version_info[0] == 3:
            key = input("\nEnter a string of 8 characters for the key: ")
        else:
            key = raw_input("\nEnter a string of 8 characters for the key: ")
        if len(key) != 8:
            print("\nKey generation needs 8 characters exactly.  Try again.\n")
            continue
        else:
            break
    key = BitVector(textstring = key)
    key = key.permute(key_permutation_1)
    return key

encryption_key = get_encryption_key()
round_keys = generate_round_keys(encryption_key)
print("\nHere are the 16 round keys:\n")
for round_key in round_keys:
    print(round_key)
```

3.4: WHAT MAKES DES A STRONG CIPHER (TO THE EXTENT IT IS A STRONG CIPHER)

- The substitution step is very effective as far as **diffusion** is concerned. It has been shown that if you change just one bit of the 64-bit input data block, on the average that alters 34 bits of the ciphertext block.

- The manner in which the round keys are generated from the encryption key is also very effective as far as **confusion** is concerned. It has been shown that if you change just one bit of the encryption key, on the average that changes 35 bits of the ciphertext.

- Both effects mentioned above are referred to as the **avalanche effect**.

- And, of course, the 56-bit encryption key means a key space of size $2^{56} \approx 7.2 \times 10^{16}$.

- Assuming that, on the average, you'd need to try half the keys in a brute-force attack, a machine able to process 1000 keys per microsecond would need roughly 13 months to break the code. However, a parallel-processing machine trying 1 million keys simultaneously would need only about 10 hours. **(EFF took three days on a specially architectured machine to break the code.)**

- The official document that presents the DES standard can be found at:

`http://www.itl.nist.gov/fipspubs/fip46-2.htm`

3.5: HOMEWORK PROBLEMS

1. A text file named `myfile.txt` that you created with a run-of-the-mill editor contains just the following word:

   ```
   hello
   ```

   If you examine this file with a command like

   ```
   hexdump  -C  myfile.txt
   ```

   you are *likely* to see the following bytes (in hex) in the file:

   ```
   68  65  6C  6C  6F  0A
   ```

   Let's now try to encrypt the contents of this text file with a 4-bit block cipher whose codebook contains the following entries:

   ```
   6, 0, 13, 4, 3, 1, 14, 8, 7, 12, 9, 15, 5, 2, 11, 10
   ```

   Let's say that I write the encrypted output into a different file and then examine this new file with the '`hexdump -C`' command. What will I see in the encrypted file?

2. In general, in a block cipher, we replace N bits from the plaintext with N bits of ciphertext. What defines an ideal block cipher?

3. Whereas it is true that the relationship between the input and the output is completely random for an ideal block cipher, it must nevertheless be invertible for decryption to work. That implies that the mapping between the input blocks and the output blocks must be one-to-one. If we had to express this mapping in the form of a table lookup, what will be the size of the table?

4. What would be the encryption key for an ideal block cipher?

5. What makes ideal block ciphers impractical?

6. What do we mean by a "Feistel Structure for Block Ciphers"?

7. Are there any constraints on the Feistel function F in a Feistel structure?

8. Explain the concepts of diffusion and confusion as used in DES.

9. If we have all the freedom in the world for choosing the Feistel function F, how should we specify it?

10. How does the permutation/expansion step in DES enhance diffusion? This is the step in which we expand by permutation and repetition the 32-bit half-block into a 48-bit half-block

11. DES encryption was broken in 1999. Why do you think that happened?

12. Since DES was cracked, does that make this an unimportant cipher?

13. **Programming Assignment 1:**

    Write a Perl or Python script that implements the full DES. Use the S-boxes that are specified for the DES standard (See Section 3.3.3). Make sure you implement all of the key generation steps outlined in Section 3.3.5. For the encryption key, your script should prompt the user for a keyboard entry that consists of at least 8 printable ASCII characters. (You may choose to either use the first seven or the last seven bits of each character byte for the 56-bit key you need for DES.)

    What makes this homework not as difficult as you think is that once you write the code that carries out one round of processing, you basically use the same code in a loop for the whole encryption chain and the decryption chain. Obviously, you will have to reverse the order in which the round keys are used for the decryption chain.

    Although you are free to write your own code from scratch, here are some recommendations: If using Python, you might want to start with the my **BitVector** class. To help you get started with the Python implementation, please see the `hw2_starter.py`

file. If using Perl, use my `Algorithm::BitVector` module from `www.cpan.org`. It is a popular Perl module for manipulating bit arrays. It is also well documented. To help you get started with the Perl implementation, please see the `hw2_starter.pl` file. You can download both these starter files through the code archive for Lecture 3.

14. **Programming Assignment 2:**

Now modify the implementation you created for the previous homework by filling the 4×16 tables for the S-boxes with randomly generated integers. Obviously, each randomly generated entry will have to be between 0 and 15, both ends inclusive. Calculate the avalanche effect for this implementation of DES and compare it with the same effect for your previous implementation. (See Section 3.3.1 for the avalanche effect.)