Lecture 9: Using Block and Stream Ciphers for Secure Wired and WiFi Communications

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

February 11, 2016

9:19am

Goals:

- To present 2DES and its vulnerability to the meet-in-the-middle attack
- To present two-key 3DES and three-key 3DES
- To present the five different modes in which a block cipher can be used in practical systems for secure communications
- To discuss stream ciphers and to review RC4 stream cipher algorithm
- To review problems with the WEP protocol for home wireless networks
- To review the Klein and PTW attacks on WEP
- Using `aircrack-ng` to crack a WEP key

CONTENTS

9.1: MULTIPLE ENCRYPTIONS WITH DES FOR A MORE SECURE CIPHER

- As you already know, the DES cryptographic system was shown to not be very secure about 10 years ago.

- We can obviously use AES cryptography that is designed to be extremely secure, but the world of commerce and finance does not want to give up on DES that quickly (because of all the investment that has already been in DES-related software and hardware).

- So that raises questions like: How about a cryptographic system that carries out repeated encryptions with DES? Would that be more secure?

- We will now show that whereas double DES may not be that much more secure than regular DES, we can expect triple DES to be very secure.

9.2: DOUBLE DES

- The simplest form of **multiple encryptions with DES is the double DES** that has two DES-based encryption stages **using two different keys**.

- Let's say that P represents a 64-byte block of plaintext. Let E represent the process of encryption that transforms a plaintext block into a ciphertext block. Let's use two 56-byte encryption keys K_1 and K_2 for a double application of DES to the plaintext. Let C represent the resulting block of ciphertext. We have

$$C = E(K_2,\ E(K_1,\ P))$$

$$P = D(K_1,\ D(K_2,\ C))$$

  where D represents the process of decryption.

- With two keys, each of length 56 bits, double DES in effect uses a 112 bit key. One would think that this would result in a dramatic increase in the cryptographic strength of the cipher — *at* *least against the brute-force attacks to which the regular DES is so vulnerable.* Recall that in a brute force attack, you try every possible key to break the code. We will argue in Section 9.2.2 that this belief is not well founded. But first, in the next subsection, let's talk about whether double DES can be thought of as a variation on the regular DES.

9.2.1: Can a Double-DES (2DES) Plaintext-to-Ciphertext Mapping be Equivalent to a Single-DES Mapping?

- Since the plaintext-to-ciphertext mapping must be one-one, the mapping created by a single application of DES encryption can be thought of as a specific permutation of the 2^{64} different possible integer values for a plaintext block. Since a permutation of a permutation is still a permutation, the following relationship between the two keys K_1 and K_2 of 2DES and some single key K_3 is obviously a theoretical possibility.

$$E(K_2,\ E(K_1,\ P)) \quad = \quad E(K_3,\ P)$$

  With such a relationship, the whole point of using 2DES to get around the weakness of DES would be lost, since in that case 2DES would be no stronger than regular DES. [Not only that, one could extend this argument to state that any number of multiple encryptions of plaintext would amount to a single encryption of regular DES. Therefore, a cipher consisting of three applications of DES encryption, as in 3DES, would be no stronger than regular DES.]

- If we said that 2DES with the two keys (K_1, K_2) is equivalent to a single application of DES with some key K_3, that would be tantamount to claiming that the set of permutations achieved with different possible 56-bit DES encryptions is closed. In other words, we would be saying that the set of permutations corresponding to DES ecryption forms a group.

- However, as it turns out, the set of permutations corresponding to DES encryptions/decryptions does not constitute a group. [For proof, see the paper entitled "DES is not a group," by Keith Campbell and Michael Wiener, that appeared in *Advances in Cryptology*, 1993. The following argument is important to that proof: The set of all possible permutations over 64-bit words is of size $2^{64}!$ as explained later in this section. This set obviously forms a group in which the group operator is that of composition-of-two-permutations (as explained in Section 4.2.2 of Lecture 4) and the group identity element is the identity permutation (meaning when each 64-bit pattern of plaintext maps to itself in the ciphertext). Now let's consider the subgroup of this group that is generated by all encryption/decryption permutations of 64-bit words that correspond to DES with 56-bit keys. The word "generated" is important here, since it implies that the subgroup will contain all permutations that are returned by applying the composition operator to any two permutations. (That's because, being a subgroup, it must be closed under the group operator.) It has been shown by Don Coppersmith that the lower bound on the size of this subgroup exceeds 2^{57}, which is the set all possible permutations that can be generated by the 56 bits of DES through encryption *and* decryption. This implies that the permutation produced by 2DES (or by, say, 3DES) is not guaranteed to belong to the set of size 2^{57} that corresponds to a single application of DES. Campbell and Weiner have estimated that the size of this subgroup is lower-bounded by 10^{2499}. The very large size of the subgroup has the following implications: Even though the subgroup being larger than 2^{57} in size does not preclude that for some choice of K_1 and K_2, 2DES would be equivalent to single DES for some K_3, the probability of finding such a triple (K_1, K_2, K_3) by searching only through the permutations created by the 56-bit DES keys is negligibly small.]

- Let's now establish why for 64-bit block encryption the total number of all possible plaintext-to-ciphertext mappings is the very large number $2^{64}!$.

- Consider 4-bit blocks. Every key gives us a unique mapping between the 16 possible words at the input and the 16 possible words at the output.

- Every mapping between the input words and the output words must amount to a **permutation** of the input words. This is necessitated by the fact that any mapping between the plaintext words and the ciphertext words must be 1-1, since otherwise decryption would not be possible.

- To understand what I mean by a **mapping** between the input words and the output words being a **permutation**, let's continue with our block size of 4 bits. Figure 1 shows one possible mapping between the 16 different input words that you can have with 4 bits and the output words. The 16 output words constitute one permutation of the 16 input words. The total number of permutations of 16 input words is 16!. [When you are looking at N *different* objects in a sequence, a permutation corresponds to the N objects appearing in a specific order. There are $N!$ ways of ordering such a sequence. Consider the case when $N = 3$ and when the objects are a, b, and c. The six different ways of arranging these objects in a sequence are abc, acb, bac, bca, cab, and cba.]

- So with a block size of 4 bits, we have a maximum of 16! mappings between the input words and the output words. In other words, we have $2^4!$ mappings when block size is 4 bits. When we select a key for encryption, we use one of these $2^4!$ mappings.

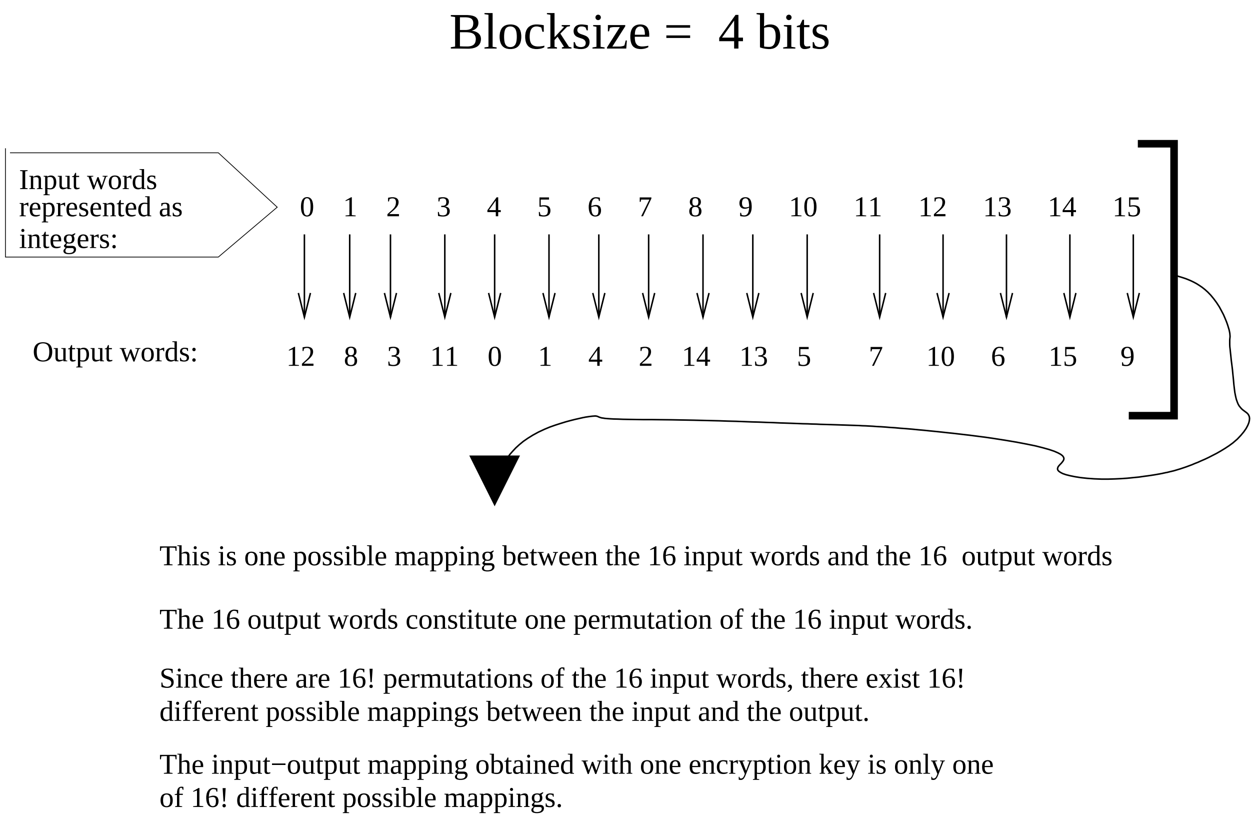

Blocksize = 4 bits
Input words represented as integers:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Output words:
12 8 3 11 0 1 4 2 14 13 5 7 10 6 15 9
This is one possible mapping between the 16 input words and the 16 output words
The 16 output words constitute one permutation of the 16 input words.
Since there are 16! permutations of the 16 input words, there exist 16! different possible mappings between the input and the output.
The input−output mapping obtained with one encryption key is only one of 16! different possible mappings.

Figure 1: *One possible mapping between the 16 different possible input words and the 16 different possible output words for a 4-bit block cipher.* (*This figure is from Lecture 9 of "Computer and Network Security" by Avi Kak*)

- Let's now extend the above argument to the case when the block size is 64 bits.

- As before, each encryption key gives us one mapping between the input 64-bit words and the output 64-bit words. Since there are 2^{64} possible words, each mapping is a relationship between the 2^{64} different possible words at the input and equal number of such words at the output.

- Since each mapping can be thought of as a permutation of the 2^{64} possible words at the input, we have a maximum of $2^{64}!$ possible mappings between the input words and the output words.

$$\begin{aligned} \left(2^{64}\right)! &= 10^{347380000000000000000} \\ &> \left(10^{10^{20}}\right) \end{aligned}$$

- Now with a key size of 56 bits, we have a total of 2^{56} different keys. Each key corresponds to one of the $2^{64}!$ different possible mappings. The number 2^{56} is upperbounded by 10^{17}.

9.2.2: Vulnerability of Double DES to the Meet-in-the-Middle Attack

- Any double block cipher, that is a cipher that carries out double encryption of the plaintext using two different keys in order to increase the cryptographic strength of the cipher, is open to what is known as the **meet-in-the-middle attack**.

- To explain the meet-in-the-middle attack, let's revisit the relationship between the plaintext P and the ciphertext C for double DES:

$$C \quad = \quad E(K_2,\ E(K_1,\ P))$$

$$P \quad = \quad D(K_1,\ D(K_2,\ C))$$

  where K_1 and K_2 are the two 56-bit keys used in the two stages of encryption.

- Let's say that an attacker has available to him/her a plaintext-ciphertext pair (P, C). From the perspective of the attacker, there exists an X such that

$$X \quad = \quad E(K_1,\ P) \quad = \quad D(K_2, C)$$

- In order to mount the attack, the attacker creates a **sorted** table of all possible value for X for a given P by trying all possible 2^{56} keys. This table will have 2^{56} entries. We will refer to this table as T_E. [The sorting can be according to the integer values of the keys.]

- The attacker also creates another sorted table of all possible X by decrypting C using every one of the 2^{56} keys. This table also has 2^{56} entries. Let's call this table T_D.

- The tables T_E and T_D are shown in Figure 2.

- Now the question is: How many of the X entries in T_E are likely to be the same as the X entries in T_D? It would obviously suit the attacker if there was a single matching entry in the T_E and T_D tables. That is, the attacker's job would be done if only one X entry in T_E were to be the same as an X entry in T_D, the entry corresponding to the actual keys K_1 and K_2 used for generating C from P. But, as we will see, in general the number of matches will be very large. So we will refer to this count as the number of **false alarms**.

- As Figure 2 shows, we need to make a total of 2^{112} comparisons in order to figure out which entries in the tables are the same. But these comparisons involve only 2^{64} different possible values for X. (Recall that X is a 64-bit word.) Then it must be case that that

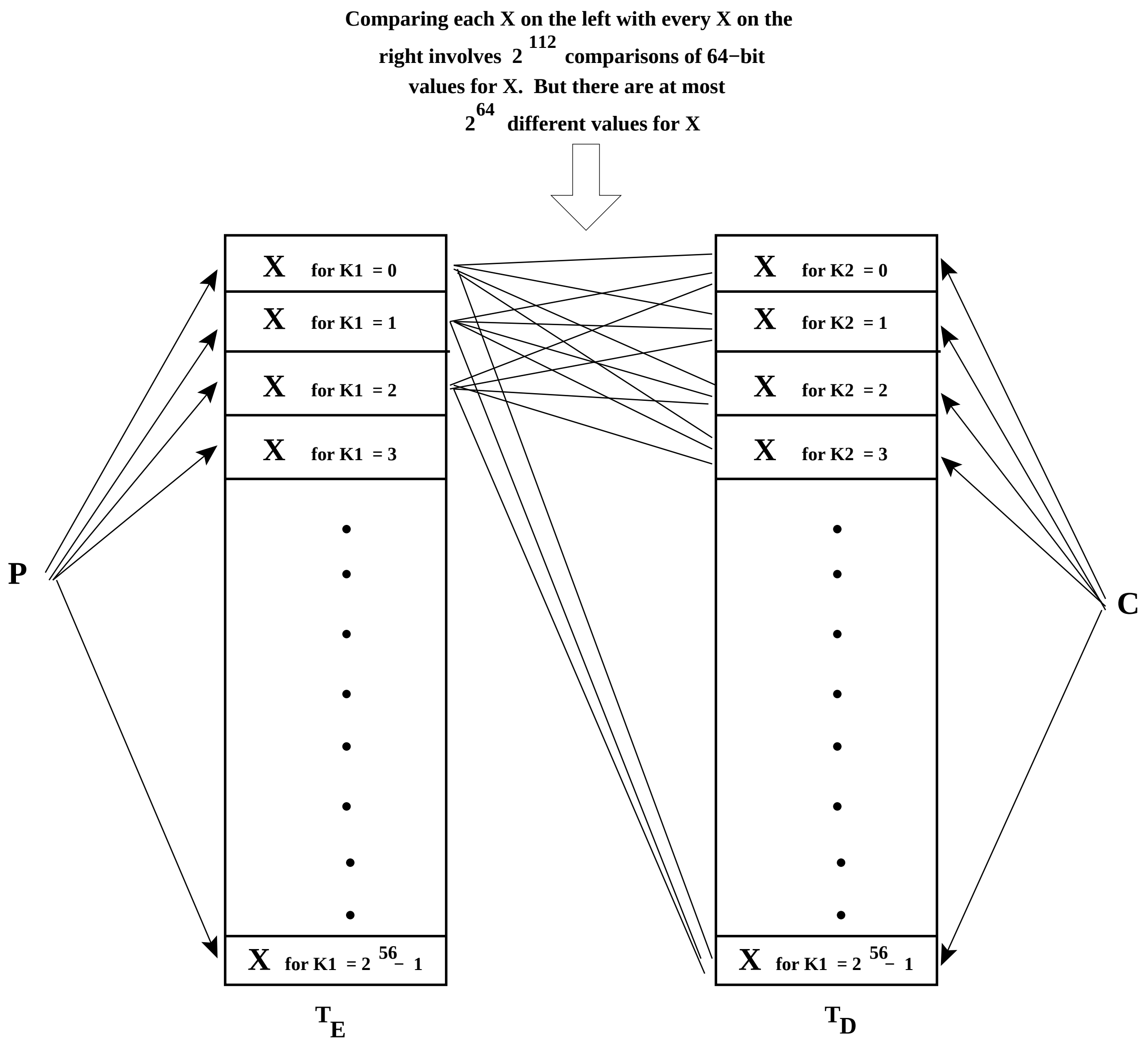

Comparing each X on the left with every X on the right involves 2^112 comparisons of 64−bit values for X. But there are at most 2^64 different values for X
X for K1 = 0
X for K1 = 1
X for K1 = 2
X for K1 = 3
X for K1 = 2^56 − 1
X for K2 = 0
X for K2 = 1
X for K2 = 2
X for K2 = 3
X for K1 = 2^56 − 1
P
C
T_E
T_D

Figure 2: *In a meet-in-the-middle attack on the 2DES cipher, an adversary uses a given plaintext-ciphertext pair (P, C) to narrow down the possible values for the two keys K_1 and K_2.* (*This figure is from Lecture 9 of "Computer and Network Security" by Avi Kak*)

$$\frac{2^{112}}{2^{64}} = 2^{48}$$

of the comparisons must involve identical values in the two tables. [Let's say you want to make a 1000 comparisons of the values of a variable under the assumption the variable can take only two values. On the average, 500 of the comparisons will involve identical values. Now consider the case when the variable can take only three values. Now a third of the comparisons must involve identical values. And so on.] So we can expect 2^{48} entries in the T_E table to be the same as the entries in the T_D entries in the T_D table.

- Therefore, when we compare the 2^{56} entries of X in T_E with the 2^{56} entries of X' in T_D, on the average we are likely to run into 2^{48} false alarms.

- Now suppose the attacker has another (P', C') pair of 64-bit words available to us. This time, we will only try the 2^{48} key pairs (K_1, K_2) on which we obtained equalities when comparing the X entries in T_E with the X' entries in T_D. Let the new tables be called T'_E and T'_D.

- Now the attacker should see no redundancy at all with regard to the X values produced by the different keys for the given P' and C'. On the other hand, now the attacker will see "negative redundancy" to the tune of $2^{48}/2^{64} = 2^{-16}$. Taken practically, that implies that there will only be a single key pair (K_1, K_2) with the same X value in the tables T'_E and T'_D.

- Therefore, the matching entry in comparing T'_E with T'_D is practically guaranteed to yield the encryption keys K_1 and K_2.

- The effort required to make such a comparison is proportional to the size of the tables T_E and T_D, which is 2^{56}, which is comparable to the effort required to break the regular DES.

9.3: TRIPLE DES WITH TWO KEYS

- An obvious defense against the **meet-in-the-middle** attack is to use triple DES.

- The most straightforward way to use triple DES is to employ three stages of encryption, each with its own key:

$$C \quad = \quad E(K_3,\ E(K_2,\ E(K_1,\ P)))$$

  But this calls for 168-bit keys, which is considered to be unwieldy for many applications.

- One way to use triple DES is with just two keys as follows

$$C \quad = \quad E(K_1,\ D(K_2,\ E(K_1,\ P)))$$

  Note that one stage of **encryption** is followed by one stage of **decryption**, followed by another stage of **encryption**. *This is also referred to as EDE encryption, where EDE stands for Encrypt-Decrypt-Encrypt.*

- There is an important reason for juxtaposing a stage of decryption between two stages of encryption: it makes the triple DES system easily usable by those who are only equipped to use regular DES. This backward compatibility with regular DES can be achieved by setting $K_1 = K_2$ in triple DES.

- It is important to realize that juxtaposing a decryption stage between two encryption stages does not weaken the resulting cryptographic system in any way. Recall, decryption in DES works in exactly the same manner as encryption. So if you encrypt data with one key and try to decrypt with a different key, the final output will be still be an encrypted version of the original input. The nature of this encrypted output will not be different, from the standpoint of cryptographic strength, from the case if you use two stages of encryption.

- Triple DES with two keys is a popular alternative to regular DES.

9.3.1: Possible Ways to Attack 3DES Based on Two Keys

- It is theoretically possible to extend the **meet-in-the-middle attack** to the case of 3DES based on two keys.

- Let's go back to the encryption equation for two-key 3DES:

$$C \quad = \quad E(K_1,\ D(K_2,\ E(K_1,\ P)))$$

  We can rewrite this equation in the following form

$$\begin{aligned} A &= E(K_1,\ P) \\ B &= D(K_2,\ A) \\ C &= E(K_1,\ B) \end{aligned}$$

- If the attacker had some way of knowing the intermediate value A for a given plaintext P, breaking the 3DES cipher becomes the same as breaking 2DES with the meet-in-the-middle attack.

- In the absence of knowledge of A, the attacker can assume some arbitrary value for A and can then try to find a known $(P,\ C)$ that results in that A by using the following procedure:

Step 1: The attacker procures n pairs of $(P,\ C)$. These are arranged in a two-column table, with all the P's in one column and their corresponding C's in the other column. This table has n rows. We refer to this table as **Table I**.

Step 2: The attacker now chooses an arbitrary A. Using this A, the attacker figures out the plaintext that will result in that A for every possible key K_1:

$$P \quad = \quad D(K_1,\ A)$$

(Recall that, in encryption, A is related to P by $A\ =\ E(K_1, P)$.) If a P calculated in this manner is found to match one of the rows in Table I, for the key K_1 that yielded this match we now find B from

$$B \quad = \quad D(K_1,\ C)$$

for the C value that corresponds to the P value in Table I. This B value and its corresponding key K_1 is entered as a row in **Table II**. (Recall that, in encryption, C is related to B by $C\ =\ E(K_1,\ B)$.)

Step 3: Given all the available $(P,\ C)$ pairs, we now fill Table II with $(B,\ K_1)$ pairs where the set of K_1's **constitutes our candidate pool for the K_1 key**.

Step 4: We now sort Table II on the B values.

Step 5: In Table II constructed as above, the left column entries, meaning B's, were obtained from the available samples of ciphertext C. Recall, the other way to obtain B is by

$$B \quad = \quad D(K_2,\ A)$$

We now try, one at a time, all possible values for the K_2 key in this equation for the assumed value for A. (Obviously, there are 2^{56} possible values for K_2.) When we get a B that is in one of the rows of Table II, we have found a candidate pair $(K_1,\ K_2)$.

Step 6: The candidate pair of keys $(K_1,\ K_2)$ is tested on the remaining $(P,\ C)$ pairs. If the test fails, we try a different value for A in Step 2 and the process is repeated.

- Let's now talk about the effort involved in arriving at a correct guess for the $(K_1,\ K_2)$ pair of keys.

- For a given pair (P, C), the probability of guessing the correct intermediate A is $1/2^{64}$.

- Therefore, given the n pairs of $(P,\ C)$ values in Table I, the probability that **a particular chosen value for** A will be correct is $n/2^{64}$.

- Now we will use the following result in probability theory: the **expected number of draws required** to draw one red ball from a bin containing n red balls and $N - n$ green balls is $(N + 1)/(n + 1)$ **if the balls are NOT replaced**.

- Therefore, given the n pairs for $(P,\ C)$, the number of different possible values for A that we may have to try is given by

$$\frac{2^{64}\ +\ 1}{n\ +\ 1} \quad \approx \quad \frac{2^{64}}{n}$$

  which is roughly in agreement with the probability $n/2^{64}$ of choosing the correct value for A if we are given n pairs for $(P,\ C)$.

- Because the size of the effort involved in Step 5 is of the order of 2^{56}, the above expression implies that the running time of the attack would be of the order of

$$2^{56} \ \cdot\ \frac{2^{64}}{n} \quad = \quad 2^{120-\log n}$$

9.4: TRIPLE DES WITH THREE KEYS

- If you don't mind 168-bit keys, here is a 3-key version of a more secure cipher that is based on multiple encryptions with DES:

$$C = E(K_3,\ D(K_2,\ E(K_1,\ P)))$$

  where the decryption step in the middle is purely for the sake of backward compatibility with the regular DES, with 2DES, and with 3DES using two keys.

- When all three keys are the same, that is when $K_1 = K_2 = K_3$, 3DES with three keys become identical to regular DES.

- When $K_1 = K_3$, we have 3DES with two keys.

- Note that as with 3DES using two keys, the decryption stage in the middle does NOT reduce the cryptographic strength of 3DES with three keys. Especially since the encryption and decryption algorithms are the same in DES, decrypting with a key that is different from the key used in encryption does not bring the output any closer to the input.

- A number of internet-based applications have adopted 3DES with three keys. **These include PGP and S/MIME**. [PGP is used for email and file storage security; we will talk about it in Lecture 20. S/MIME stands for Secure-MIME and MIME stands for **Multipurpose Internet Mail Extensions**. When you attach PDF files, photos, videos, etc., with your email, they are sent as MIME objects.]

9.5: FIVE MODES OF OPERATION FOR BLOCK CIPHERS

- The discussion in this section applies to **all** block ciphers, including the AES cipher presented in Lecture 8.

- Just because a block cipher has been demonstrated to be strong (in the sense that it is not vulnerable to brute-force, meet-in-the-middle, typical statistical, and other such attacks), does not imply that it will be sufficiently secure if you are using it to transmit long messages. [By "long", we mean many times longer than the block length.] The interaction between the block-size based periodicity of such ciphers and any repetitive structures in the plaintext may still leave too many clues in the ciphertext that compromise its security.

- The goal of this section (which includes the five subsections that follow) is to present the five different modes in which any block cipher can be used. The first of these, ECB, is for using a block cipher as it is, meaning by scanning a long document one block at a time and enciphering it independently of the blocks seen before or the blocks to be seen next. As will be pointed out, this is not suitable for long messages. It is the next four modes, variations on the first, that are actually used in real-world applications for the encryption of long messages.

Electronic Code Book (ECB): This method is referred to as the Electronic Code Book method because the encryption process can be represented by a *fixed mapping* between the input blocks of plaintext and the output blocks of cipher text. So it is very similar to the code book approach of the distant past. The code book would list the ciphertext mapping for each plaintext word. For this mode to work correctly, either the message length must be an integral multiple of the block size or you must use padding so that the condition on the length is satisfied.

Cipher Block Chaining Mode (CBC): The input to the encryption algorithm is the XOR of the next block of plaintext and the previous block of ciphertext. This is obviously more secure for long segments of plaintext. However, this mode also requires that length of the plaintext message be an integral multiple of the block size. When that condition is not satisfied, the message must be suitably padded.

Cipher Feedback Mode (CFB): Whereas the CBC mode uses all of the previous ciphertext block to compute the next ciphertext block, the CFB mode uses only a fraction thereof. Also, whereas in the CBC mode the encryption system digests b bits of plaintext at a time (where b is the blocksize used by the block cipher), now the encryption system digests only $s < b$ number of plaintext bits at a time even though the encryption algorithm itself carries out a b-bits to b-bits transformation. Since s can be any number, including one byte, that makes CFB suitable as a stream cipher.

Output Feedback Mode (OFB): The basic logic here is the same as in CFB, only the nature of what gets fed from stage to stage is different. In CFB, you feed $s < b$ number of ciphertext bits from the current stage into the b-bits to b-bits transformation carried out by the next-stage encryption. But in OFB, you feed s bits from the output of the transformation itself. This mode of operation is also suitable if you want to use a block cipher as a stream cipher.

Counter Mode (CTR): Whereas the previous four modes for using a block cipher are intuitively plausible, this new mode at first seems strange and seemingly not secure. But it has been theoretically established that this mode is at least as secure as the other modes. As for CFB and OFB, an interesting property of this mode is that only the encryption algorithm is used at both the encryption end and at the decryption end. The basic idea consists of applying the encryption algorithm **not** to the plaintext directly, but to a b-bit number (and its increments modulo 2^b for successive blocks) that is chosen beforehand. The ciphertext consists of what is obtained by XORing the encryption of the number with a b-bit block of plaintext.

In Sections 9.5.1 through 9.5.5 that follow, we will examine in greater detail these five different modes for using a block cipher.

9.5.1: The Electronic Code Book Mode (ECB)

- When a block cipher is used in ECB mode, each block of plaintext is coded independently. This makes it not very secure for long segments of plaintext, especially plaintext containing repetitive information (**particularly if the nature of what is repetitive in the plaintext is known to the attacker**). Used primarily for secure transmission of short pieces of information, such as an encryption key.

- I will now demonstrate visually that when each block of a plaintext file is encrypted independently of the other blocks, the "structure" of the information in the ciphertext file can hold important clues to what is in the plaintext file.

- Shown in Figure 3(a) is a graylevel image of a rose. Figure 3(b) shows the edge-detected version of the rose in (a). [For the images that are shown, I started with a colored jpeg image of a rose that I converted to the black-and-white ppm format with the ImageMagick package using the '`convert -colorspace Gray -equalize americanpride.jpg myimage.ppm`' command, where `americanpride.jpg` is the name of the original color image and `myimage.jpg` the name of the output file for the black-and-white image. The '-equalize' option carries out histogram equaliztion of the gray levels in the output for a superior black-and-white image. Note that you need to carry out the jpeg to ppm conversion because the bytes in the jpeg format do NOT directly represent the pixel brightness values. On the other hand, after the file header, each byte in a ppm file is a grayscale value at a pixel. In other words, after the file header, the bytes in a ppm file are the raw image data. (The file header contains information regarding the size of the image, etc.) The edge-detected version of the rose was produced by the command: `convert -blur 5x2 -edge 0 myimage.ppm my_edge_image.ppm` which gives us the result shown in (b). The option '`-edge 0`' means that we want edges to be one pixel wide and the option '-blur 5x2' means that, prior to edge detection, we want the image to be smoothed by an 5×5 Gaussian operator whose variance equals 2 pixels.] When we apply DES block encryption to the data in Figure 3(b) and simply display the ciphertext bytes as image gray levels, we get what is shown in Figure 3(c). The ciphertext bytes that are displayed in Figure 3(c) were generated by the following Perl script [This script takes two command-line arguments, the name of the ppm file containing the edge image and the name of the output ppm file into which the ciphertext data will be deposited]:

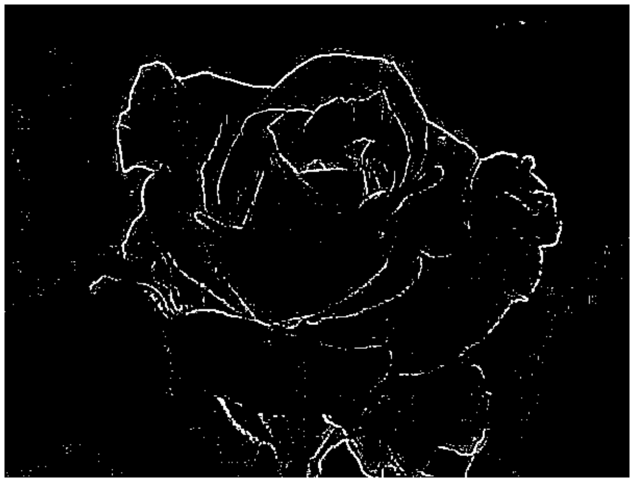
```
#!/usr/bin/env perl

##  ImageDESEcrypt.pl

##  Avi Kak
##  February 12, 2015

##  This script uses the DES algorithm in the ECB mode to encrypt an image
##  to demonstrate shortcomings of the ECB.  It is best to call this script
##  on an edge-enhanced image.

##  Call syntax:
##
##     ImageDESEncrypt.pl  input_image.ppm   output.ppm

use strict;                                                                #(A)
use warnings;
use Crypt::ECB;                                                            #(B)
use constant BLOCKSIZE => 64;                                              #(C)

die "Needs two command-line arguments for in-file and out-file"            #(D)
    unless @ARGV == 2;                                                     #(E)

my $crypt = Crypt::ECB->new;                                               #(F)
# It is important to supply the PADDING_NONE option here.  With the other
# option, PADDING_AUTO, it will padd extra 8 bytes to each block of 8 bytes
# I read and feed into the encryption function.  This padding, presumably
# all zeros, probably makes sense when you supply the entire file to the
# encrypt function all at once.
$crypt->padding(PADDING_NONE);                                             #(G)
$crypt->cipher('DES') || die $crypt->errstring;                            #(H)
```

```
$crypt->key('hello123');                                                    #(I)

open FROM, shift @ARGV or die "unable to open filename: $!";               #(J)
open TO, ">" . shift @ARGV or die "unable to open filename: $!";           #(K)
binmode( FROM );                                                            #(L)
binmode( TO );                                                              #(M)

my $encrypted = "";                                                         #(N)
my $total_bytes_read = 0;                                                   #(O)
$|++;                                                                       #(P)
while (1) {                                                                 #(Q)
    my $num_of_bytes_read = sysread( FROM, my $buff, BLOCKSIZE/8 );         #(R)
    $total_bytes_read += $num_of_bytes_read;                                #(S)
    if ($total_bytes_read < 2048) {                                         #(T)
        $encrypted .= $buff;                                                #(U)
        next;                                                               #(V)
    }
    $buff .= '0' x (BLOCKSIZE/8 - $num_of_bytes_read)
                       if ($num_of_bytes_read < BLOCKSIZE/8);               #(W)
    $encrypted .= $crypt->encrypt( $buff );                                 #(X)
    print ". " if $total_bytes_read % 2048 == 0;                            #(Y)
    last if $num_of_bytes_read < BLOCKSIZE/8;                               #(Z)
}
syswrite( TO, $encrypted );                                                 #(a)
```

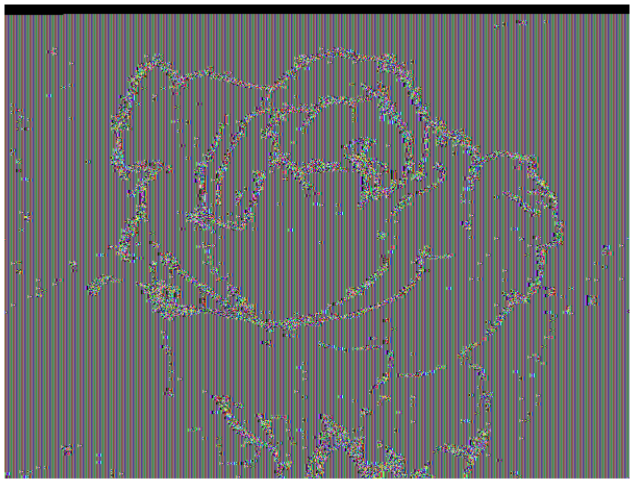
Starting in line (Q), note in the "**`while`**" loop how we do not encrypt the first 2048 bytes in the image file that is subject to encryption. These initial bytes are transfered directly to the output ciphertext file. This is done to preserve the file header so that the display program would recognize the ciphertext data as a ppm image. Also note that in the script shown above, the **`Crypt::ECB`** module is asked to use no padding and to use the DES algorithm for block encryption. It is important to turn off automatic padding, as I have done in line (G), for this demonstration to work.

- Lest you think that our being able to see the outline of the flower in the ciphertext data in Figure 3(c) may have something to do

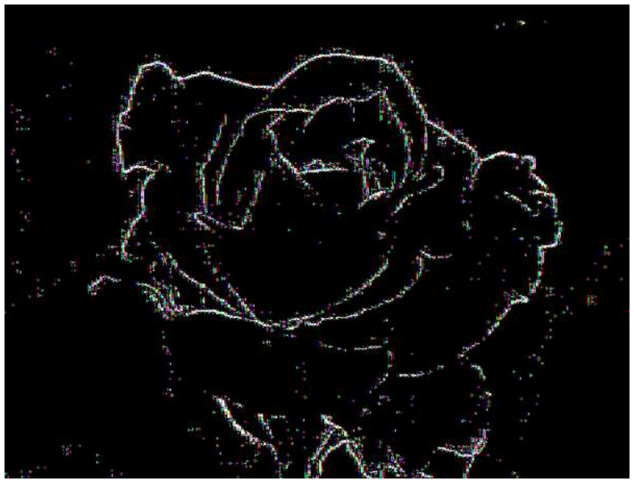
(a) rose.ppm



(b) rose_edgemap.ppm



(c) cipher_rose.ppm



(d) cipher_rose2.ppm

Figure 3: *Shown here are the security risks associated with using a block cipher without chaining. What you see in (b) is an edge image for the rose in (a). The DES-ECB encrypted version of (b) is shown in (c), whereas (d) shows the encrypted output obtained with another block cipher.* (*This figure is from Lecture 9 of "Computer and Network Security" by Avi Kak*)

with the DES algorithm, shown in Figure 3(d) is the ciphertext data obtained with a completely different approach to block encryption. Here we carry out block encryption by randomly permuting the 64 bits in each block according to a pseudorandom order specified by the encryption key. This encryption key itself is generated by randomly permuting a list consisting of the first 64 integers. In Perl, you can conveniently do that with the help of the Fisher-Yates shuffle. See the script that follows.

```
#!/usr/bin/env perl

##  ImageBlockEcrypt.pl

##  Avi Kak (February 13, 2015)

##  Each block of bits read from the image file is represented as an instance
##  of the following class:
##
##             Algorithm::BitVector
##
##  that you can download from the CPAN archive at
##
##  http://search.cpan.org/~avikak/Algorithm-BitVector-1.21/lib/Algorithm/BitVector.pm

##  The block encryption used here is based on a random permutation of the
##  bits in the source file.  For a receiving party to decrypt the
##  information, you will have to send them the key file that is created in
##  line (K).

##  Call syntax:
##
##     ImageBlockEncrypt.pl  input_image.ppm   output.ppm

use strict;
use warnings;
use Algorithm::BitVector;                                                      #(A)
use constant BLOCKSIZE => 64;                                                  #(B)

die "Needs two command-line arguments for in file and out file"                #(C)
    unless @ARGV == 2;                                                         #(D)
$|++;                                                                          #(E)

my $inputfile = shift;                                                         #(F)
open my $TO, ">" . shift @ARGV or die "unable to open filename: $!";           #(G)
```

```
# Open ‘keyfile.txt’ so that you can write the permutaiton order into the
# file (this serves as our "encryption key"):
open KEYFILE, "> keyfile.txt";                                           #(H)
my @permute_indices = 0..BLOCKSIZE-1;                                    #(I)
# Now create a random permutation of the bit positions.  We will use this
# method for encryption in this script.  If you had to represent the
# permutations as an encryption key, that would be a very long key indeed.
fisher_yates_shuffle( \@permute_indices );                               #(J)
print KEYFILE "@permute_indices";                                        #(K)
close KEYFILE;                                                           #(L)

# Let’s now start scanning the input file and encrypting it by permuting
# the bits in each block:
my $j = 0;
my $bv = Algorithm::BitVector->new( filename => $inputfile );            #(M)
while ($bv->{more_to_read}) {                                            #(N)
    print "." if $j % 1000 == 0;                                         #(O)
    my $bv_read = $bv->read_bits_from_file( BLOCKSIZE );                 #(P)
    if ($j++ < 2048) {                                                   #(Q)
        $bv_read->write_to_file( $TO );                                  #(R)
        next;
    }
    if ($bv_read->length() < BLOCKSIZE) {                                #(S)
        $bv_read->pad_from_right(BLOCKSIZE - $bv_read->length());        #(T)
    }
    my $permuted_bitvec = $bv_read->permute(\@permute_indices );         #(U)
    $permuted_bitvec->write_to_file( $TO );                              #(V)
}                                                                        #(W)
$bv->close_file_handle();                                                #(X)

sub fisher_yates_shuffle {                                               #(Y)
    my $arr =  shift;                                                    #(Z)
    my $i = @$arr;                                                       #(a)
    while (--$i) {                                                       #(b)
        my $j = int rand( $i + 1 );                                      #(c)
        @$arr[$i, $j] = @$arr[$j, $i];                                   #(d)
    }
}
```

- As you can see from the results shown, straightforward block encryption can leave too many clues in the ciphertext for an attacker. For this reason, a straightforward approach to block encryption (meaning using it in the ECB mode) is good only for short messages or messages without too much repetitive structure. In the image data that we used in our demonstration here, there was too much repetitiveness in the the background — since most of those pixels were zero — and this repetitiveness was only occasionally broken by sudden appearances of gray values at the edges.

- Another shortcoming of ECB is that the length of the plaintext message must be integral multiple of the block size. When that condition is not met, the plaintext message must be padded appropriately.

- The next three modes presented in Sections 9.5.2 through 9.5.4 provide enhanced security by making the ciphertext for any block a function of all the blocks seen previously. These modes also do **not** require that the size of the plaintext be an integral multiple of the block size.

- It is highly recommended that you apply the DES script you wrote for one of your homeworks to an image taken with your digital camera to see for yourself the results presented here.

- Shown in the rest of this section are the Python versions of the Perl scripts presented earlier. I first present the Python script that carries out DES encryption in the ECB mode.

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

##  ImageDESEcrypt.py
##  Avi Kak
##  February 11, 2016

##  This script uses the DES algorithm in the ECB mode to encrypt an image
##  to demonstrate shortcomings of the ECB.  It is best to call this script
##  on an edge-enhanced image.

##  Call syntax:
##
##     ImageDESEncrypt.py  input_image.ppm   output.ppm


import sys
from Crypto.Cipher import DES                                                  #(A)

if len(sys.argv) is not 3:                                                     #(B)
    sys.exit('''Needs two command-line arguments, one for '''
             '''the source image file and the other for the '''
             '''encrypted output file''')

BLOCKSIZE = 64                                                                 #(C)

cipher = DES.new(b'hello123', DES.MODE_ECB)                                    #(D)

FROM = open(sys.argv[1], 'rb')                                                 #(E)
TO = open(sys.argv[2], 'wb')                                                   #(F)

end_of_file = None                                                             #(G)
total_bytes_read = 0                                                           #(H)
while True:                                                                    #(I)
    bytestring = ''                                                            #(J)
    for i in range(BLOCKSIZE // 8):                                            #(K)
        byte = FROM.read(1)                                                    #(L)
        if byte == '':                                                         #(M)
            end_of_file = True                                                 #(N)
            break                                                              #(O)
        else:
            total_bytes_read += 1                                              #(P)
            bytestring += byte                                                 #(Q)
    if end_of_file:                                                            #(R)
        bytestring += '0' * (8 - total_bytes_read % 8)                         #(S)
    cipherout = cipher.encrypt(bytestring) if total_bytes_read >= 2048 else bytestring #(T)
    TO.write(cipherout)                                                        #(U)
    if end_of_file: break                                                      #(V)
    if total_bytes_read %2048 == 0:                                            #(W)
        print ".",                                                             #(Y)
        sys.stdout.flush()                                                     #(Z)
TO.close()
```

- You would call the script shown above in exactly the same way as you did for the Perl script **ImageDESEncrypt.pl** presented earlier. In other words, your call will look like

```
    ImageDESEncrypt.py    your_edge_enhanced_image.ppm    output_image.ppm
```

- Finally, here is **ImageBlockEncrypt.py** as the Python version of the Perl script **ImageBlockEncrypt.pl** presented earlier:

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

##  ImageBlockEcrypt.py

##  Avi Kak (February 11, 2016)

##  Each block of bits read from the image file is represented as an instance of the
##  Python BitVector class.

##  The block encryption used here is based on a random permutation of the bits in
##  the source file.  For a receiving party to decrypt the information, you will have
##  to send them the key file that is created in line (K).

##  Call syntax:
##
##     ImageBlockEncrypt.py  input_image.ppm   output.ppm

import sys
import random
from BitVector import *                                                         #(A)

if len(sys.argv) is not 3:                                                      #(B)
    sys.exit('''Needs two command-line arguments, one for '''
     '''the source image file and the other for the '''
     '''encrypted output file''')

BLOCKSIZE = 64                                                                  #(C)
inputfile = sys.argv[1]                                                         #(D)
TO = open(sys.argv[2], 'w')                                                     #(E)

# Open `keyfile.txt' so that you can write the permutaiton order into the
# file (this serves as our "encryption key"):
KEYFILE = open("keyfile.txt", 'w')                                              #(F)
permuted_indices = range(BLOCKSIZE)                                             #(G)
# Now create a random permutation of the bit positions.  We will use this
# method for encryption in this script.  If you had to represent the
# permutations as an encryption key, that would be a very long key indeed.
random.shuffle(permuted_indices)                                                #(H)
```

```
KEYFILE.write(str(permuted_indices))                                                  #(I)
KEYFILE.close()                                                                       #(J)

# Let's now start scanning the input file and encrypting it by permuting
# the bits in each block:
j = 0                                                                                 #(K)
bv = BitVector( filename = inputfile )                                                #(L)
while bv.more_to_read:                                                                #(M)
    if j %1000 == 0:                                                                  #(N)
        print ".",                                                                    #(O)
        sys.stdout.flush()                                                            #(P)
    bv_read = bv.read_bits_from_file( BLOCKSIZE )                                     #(Q)
    j += 1                                                                            #(R)
    if j < 2048:                                                                      #(S)
        bv_read.write_to_file( TO )                                                   #(T)
        continue                                                                      #(U)
    if bv_read.length() < BLOCKSIZE:                                                  #(V)
        bv_read.pad_from_right(BLOCKSIZE - bv_read.length())                          #(W)
    permuted_bitvec = bv_read.permute( permuted_indices )                             #(X)
    permuted_bitvec.write_to_file( TO )                                               #(Y)
bv.close_file_object();                                                               #(Z)
TO.close()
```

- The call syntax for the script shown above is the same as what you saw earlier:

```
    ImageBlockEncrypt.py    your_edge_enhanced_image.ppm    output_image.ppm
```

9.5.2: The Cipher Block Chaining Mode (CBC)

- To overcome the security deficiency of the ECB mode, the input to the encryption algorithm consists of the XOR of the plaintext block and the ciphertext produced from the previous plaintext block. See Figure 4.

- This makes it more difficult for a cryptanalyst to break the code using strategies that look for patterns in the ciphertext, patterns that may correspond to the known structure of the plaintext.

- To get started, the chaining scheme shown in Figure 4 obviously needs what is known as the **initialization vector** for the first invocation of the encryption algorithm.

- The initialization vector, denoted IV, is sent separately as a short message using the ECB mode.

- With this chaining scheme, the ciphertext block for any given plaintext block becomes a function of all the previous ciphertext blocks.

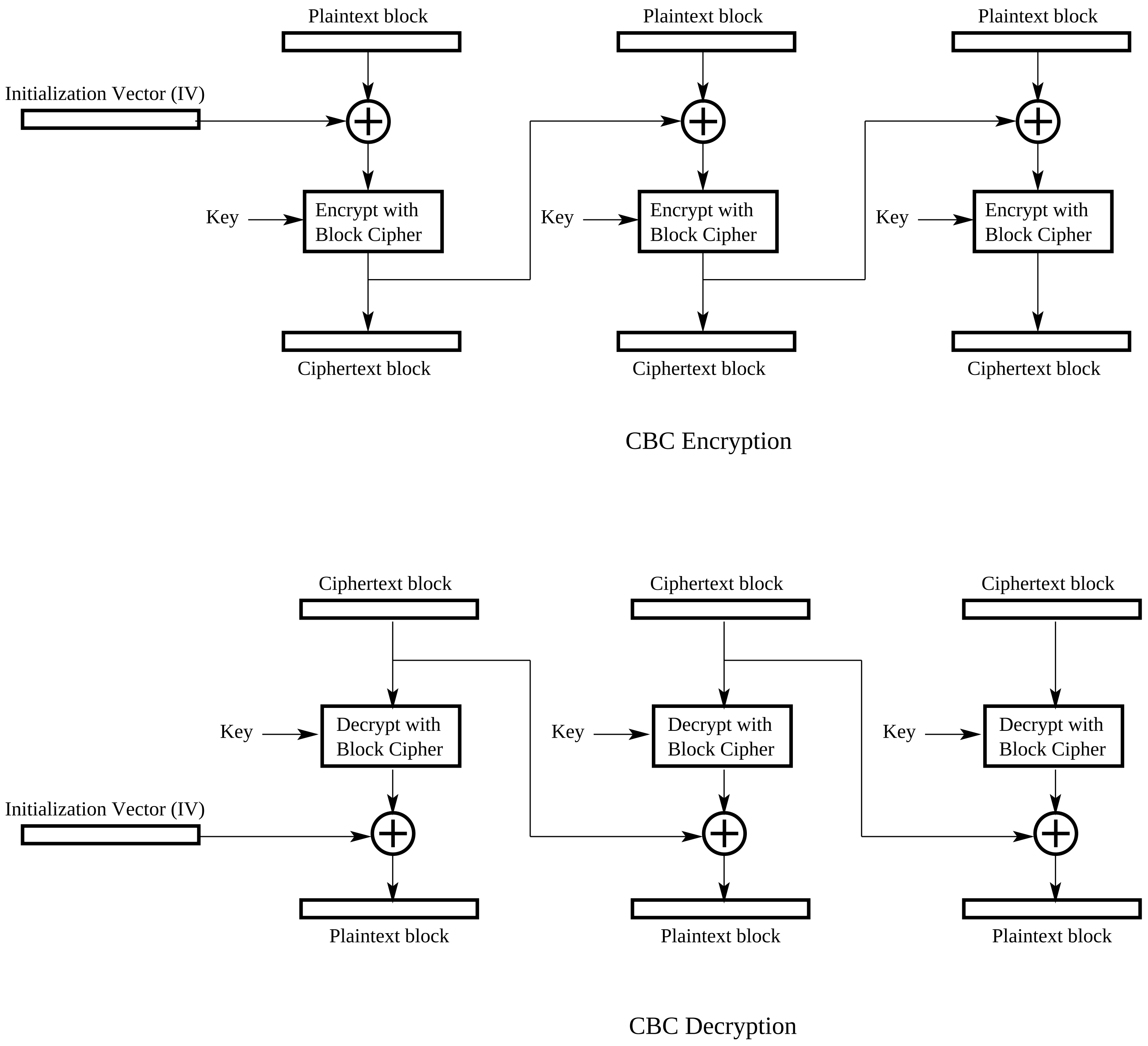


Figure 4: *The Cipher Block Chaining Mode for using a block cipher.* (*This figure is from Lecture 9 of "Computer and Network Security" by Avi Kak*)

9.5.3: The Cipher Feedback Mode (CFB)

- This approach. illustrated in Figure 5, allows a block cipher to be used as a **stream cipher**. [With a block cipher, if the length of the message is not an integral number of blocks, you must pad the message. It is not necessary to do so with a stream cipher.]

- This mode works as follows:

  – Start with an **initialization vector**, IV, of the same size as the blocksize expected by the block cipher. The IV is stored in shift register for reasons that will shortly be clear.

  – Encrypt the IV with the block cipher encryption algorithm.

  – Retain only one byte from the output of the encryption algorithm. Let this be the most significant byte. Discard the rest of the output.

  – XOR the byte retained with the byte of the plaintext that needs to be transmitted. Transmit the output byte produced.

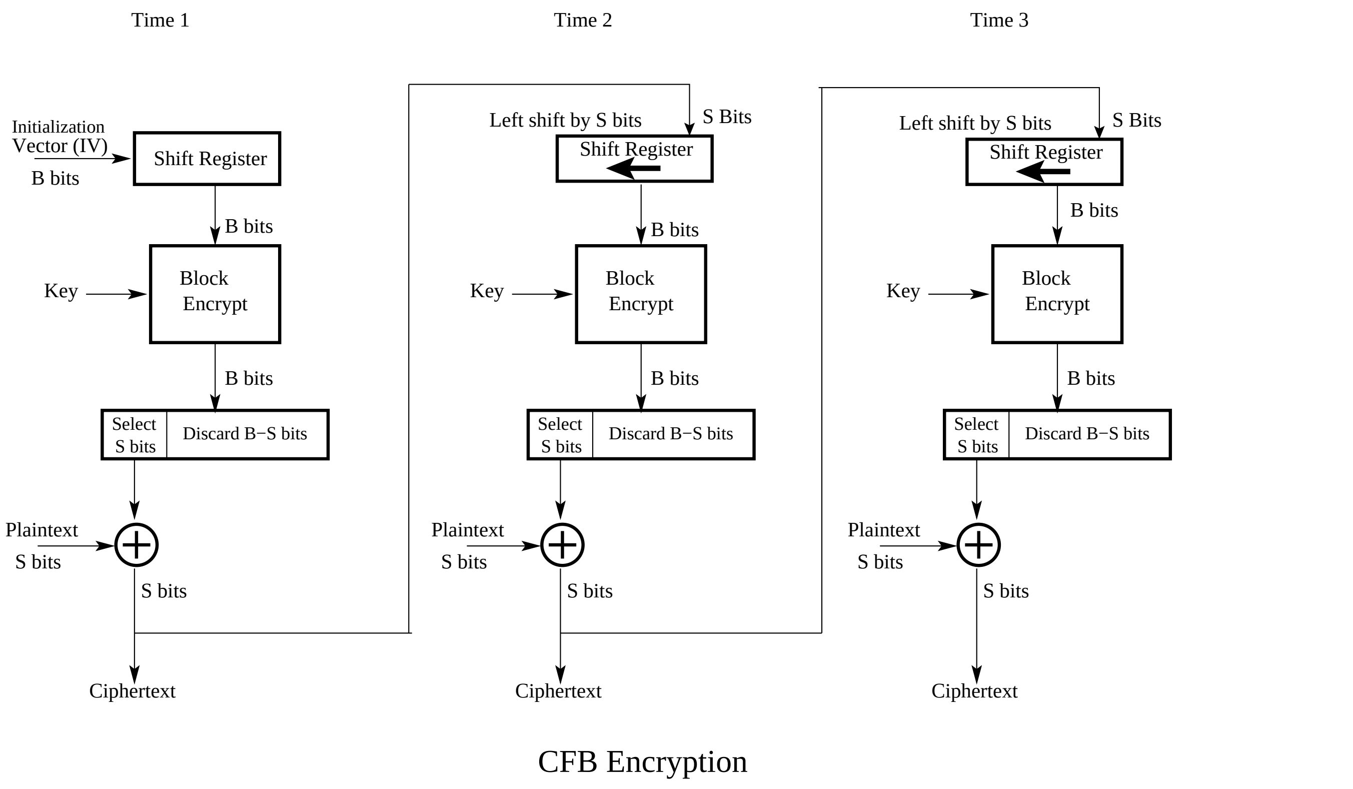
  – Shift the IV one byte to the left (discarding the leftmost byte) and insert the ciphertext byte produced by the previous step as the rightmost byte. So the new IV is still of the same length as the block size expected by the encryption algorithm.

– Go back to the step "Encrypt the IV with the block cipher encryption algorithm".

- Figure 5 shows these steps on a recurring basis for both encryption and decryption. The figure is slightly more general than the description above because it assumes that you want the unit of transmission to be s bits, as opposed to 1 byte. But it is typically the case that $s = 8$.

- A most important thing to note about the scheme in Figure 5 is that only the encryption algorithm is used in both encryption and decryption. This can be an important implementation-level detail for those block ciphers for which the encryption and the decryption algorithms are significantly different. AES is a case in point.

- Note that the ciphertext byte produced for any plaintext byte depends on all the previous plaintext bytes in the CFB mode.



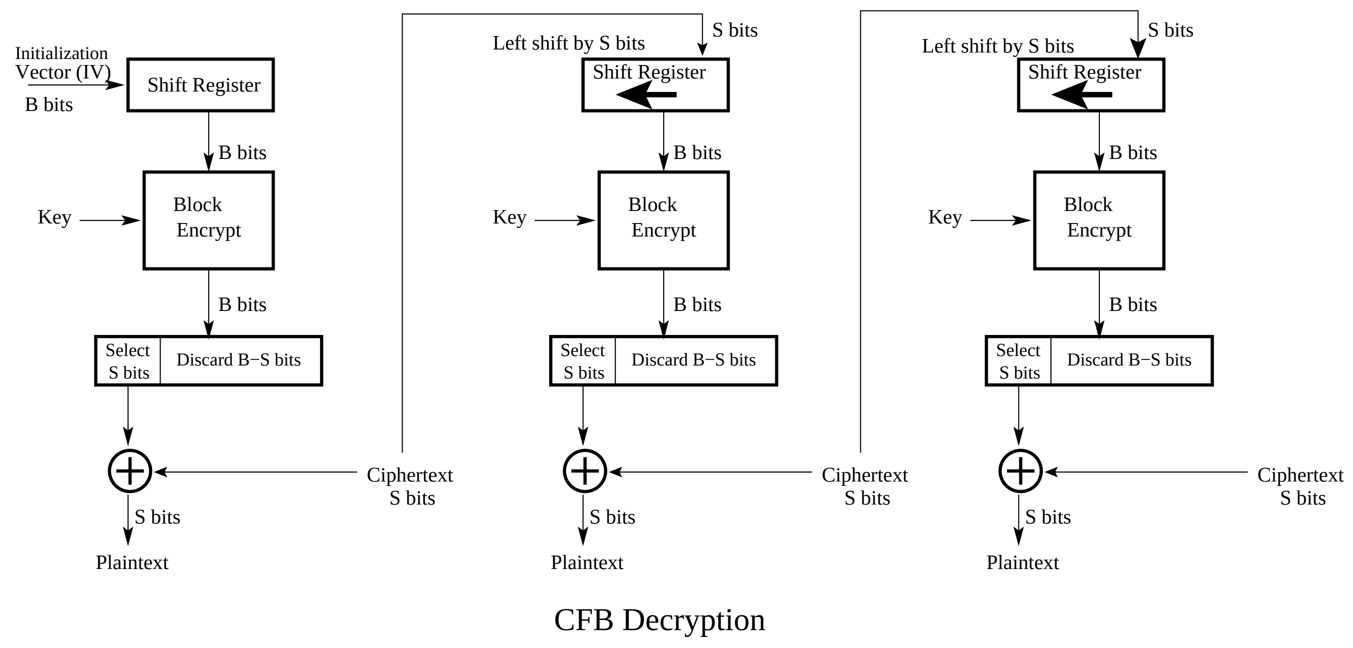


Figure 5: *The Cipher Feedback Mode for using a block cipher.* (*This figure is from Lecture 9 of "Computer and Network Security" by Avi Kak*)

9.5.4: The Output Feedback Mode (OFB)

- Very similar to the CFB mode. Therefore, this scheme can also be used as a stream cipher.

- The only difference between CFB and OFB is that, as shown in Figure 6, now we feed back one byte (the most significant byte) from the output of the block cipher encryption algorithm, as opposed to feeding back the actual ciphertext byte. This, as further explained below, makes OFB more resistant to transmission bit errors.

- Considering CFB, let's say that you have encrypted and transmitted the first byte of plaintext. Now suppose this byte is received with a one or more bit errors. In addition to producing an erroneous decryption for the first byte, that error will also propagate to downstream decryptions because the received ciphertext byte is also fed back into the decryption of the next byte.

- On the other hand, what is fed back in OFB is completely locally generated at the receiver. That is, the information that is fed back is not exposed to the possibility of transmission errors in OFB.

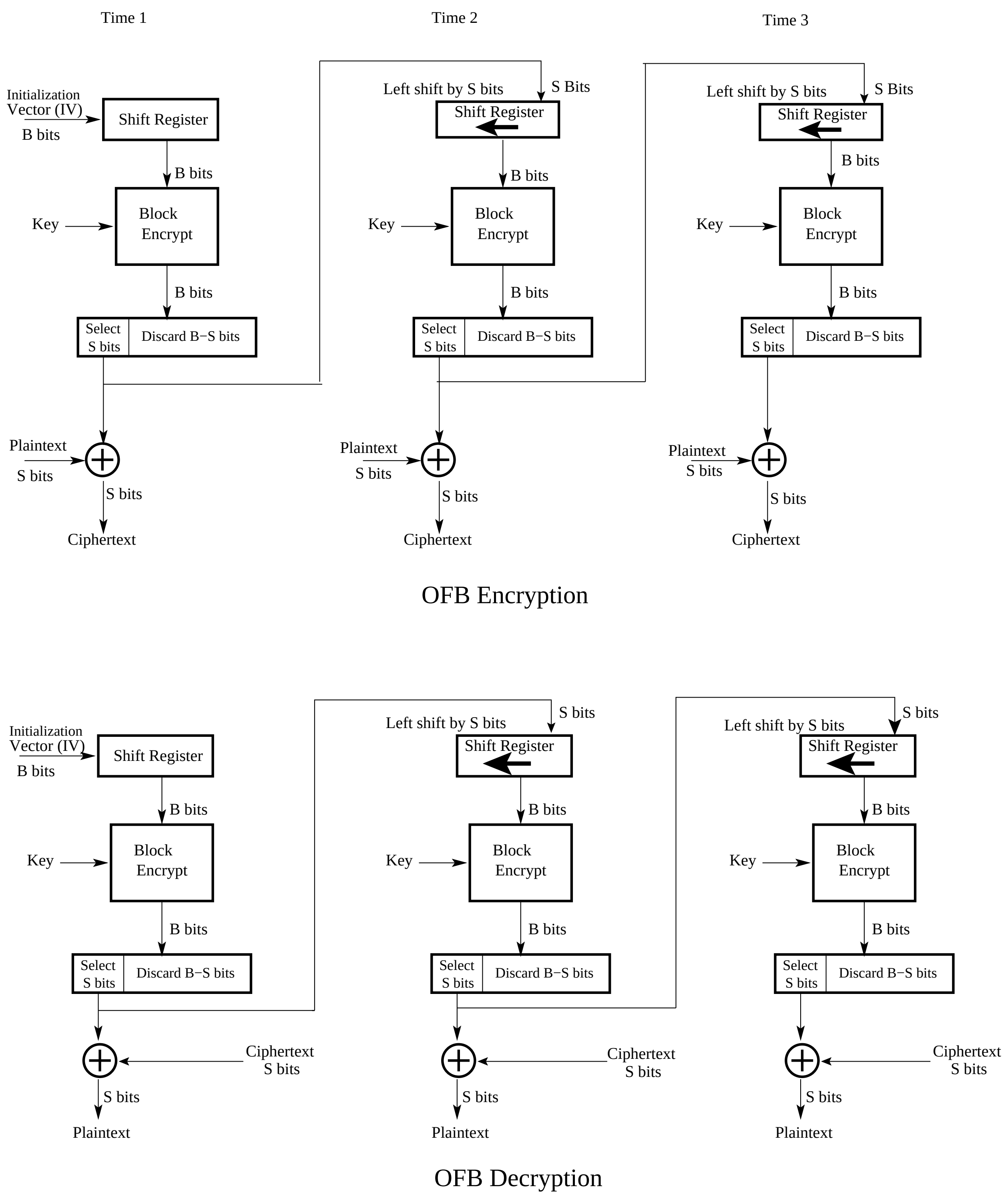


Figure 6: *The Output Feedback Mode for using a block cipher.* (*This figure is from Lecture 9 of "Computer and Network Security" by Avi Kak*)

9.5.5: The Counter Mode (CTR)

- Whereas the previous two modes, CFB and OFB, are intended to use a block cipher as a stream cipher, the counter mode (CTR) retains the pure block structure relationship between the plaintext and ciphertext.

- In other words, for each b-bit input plaintext block, the scheme produces an b-bit ciphertext block. Furthermore, the block cipher encryption algorithm that is used carries out a b-bits to b-bits transformation.

- In CFB and OFB, on the other hand, whereas the block-cipher encryption algorithm did carry out a b-bits to b-bits transformation, only s bits of plaintext, with $s < b$, were converted into s bits of ciphertext at one time. Moreover, s is typically 8 for the 8 bits of a byte in CFB and OFB.

- As shown in Figure 7 (and as is also true for the OFB mode, but not for the CFB mode), no part of the plaintext is directly exposed to the block encryption algorithm in the CTR mode. The encryption algorithm encrypts only a b-bit integer **produced by the counter**. What is transmitted is the XOR of the encryption of the integer and the b bits of the plaintext.

- For the counter value, we start with some number for the first plaintext block and then increment this value modulo 2^b from block to block, as shown in Figure 7.

- Note that, as shown in Figure 7, **only the forward encryption algorithm** is used for both encryption and decryption. (This is of significance for block ciphers for which the encryption algorithm differs substantially from the decryption algorithm. AES is a case in point.) (This property of CTR is also true for CFB and OFB modes.)

- Here are some advantages of the CTR mode for using a block cipher:

  - Fast encryption and decryption. If memory is not a constraint, we can precompute the encryptions for as many counter values as needed. Then, at the transmit time, we only have to XOR the plaintext blocks with the pre-computed b-bit blocks. The same applies to fast decryption.

  - It has been shown that the CTR is at least as secure as the other four modes for using block ciphers.

  - Because there is no block-to-block feedback, the algorithm is highly amenable to implementation on parallel machines. For the same reason, any block can be decrypted with random access.

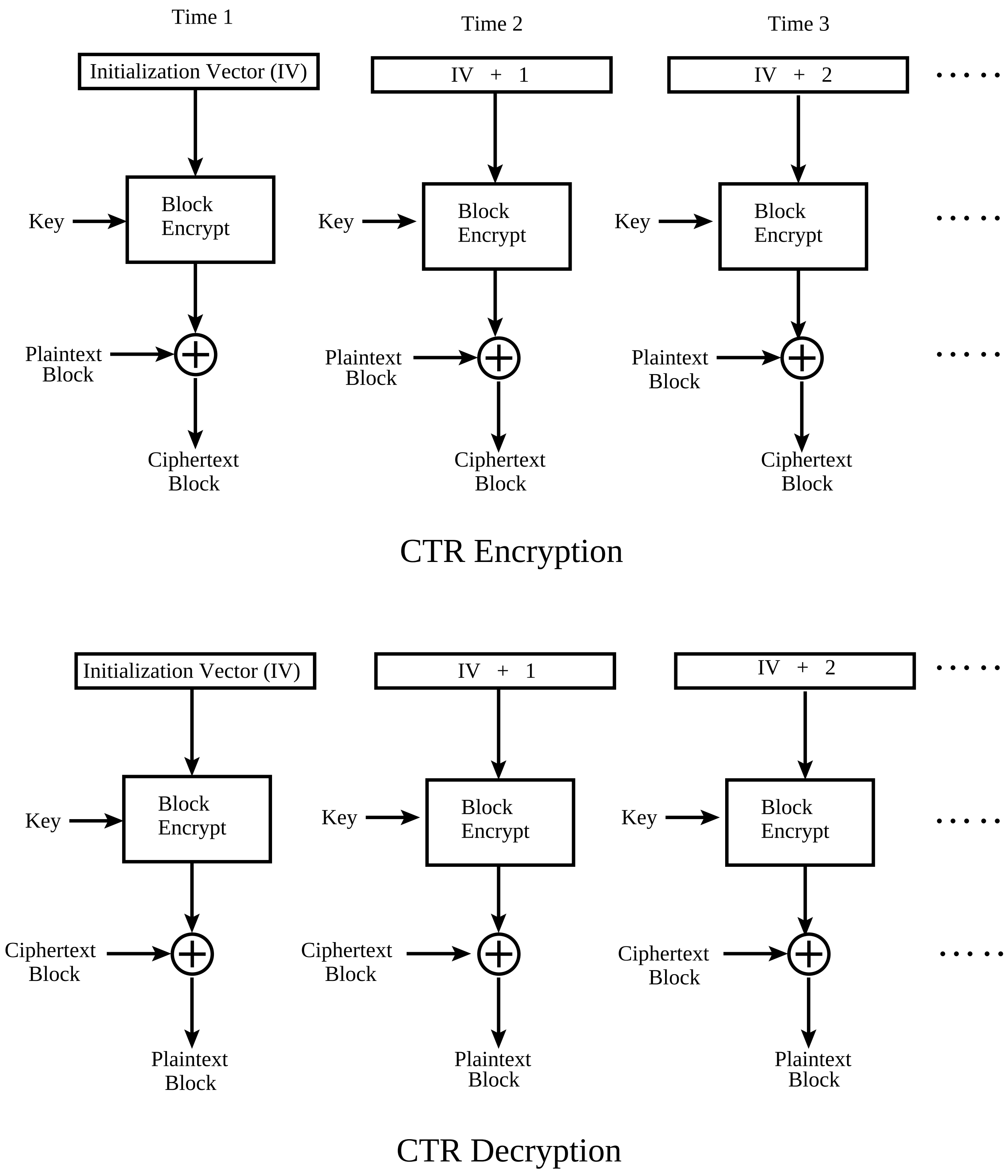


Figure 7: *The Counter Mode for using a block cipher.* (*This figure is from Lecture 9 of "Computer and Network Security" by Avi Kak*)

9.6: STREAM CIPHERS

- Previously we showed how a block cipher, when used in the CFB and OFB modes, can be deployed as a stream cipher. We will now focus on ciphers that are designed explicitly to work as stream ciphers. As you already know, a typical stream cipher encrypts plaintext one byte at a time.

- The main processing step in a true stream cipher is the generation of a **stream of pseudorandom bytes** that depend on the encryption key.

- As a new byte of plaintext shows up for encryption, a new byte of the pseudorandom stream also becomes available at the same time and this happens on a continuous basis.

- Obviously, each different encryption key will result in a different stream of pseudorandom bytes. But for a given encryption key, the stream of pseudorandom bytes will be the same at the both the encryption end and the decryption end of a data link.

- Encryption itself is as simple as it can be. You just XOR the byte from the pseudorandom stream with the plaintext byte to get the encrypted byte.

- You generate the same pseudorandom byte stream for decryption. The decryption itself consists of XORing the received byte with the pseudorandom byte.

- The encryption is shown in the left half and the decryption in the right half of Figure 8.

- For a stream cipher to be secure, the pseudorandom sequence of bytes should have as long a period as possible. Note that every pseudorandom number generator produces a seemingly random sequence that **eventually** repeats. The longer the period, the more difficult it is to break the cipher.

- Within the periodicity limitations of a pseudorandom byte sequence generator, the sequence should be as random as possible. From a statistical point, that means that all of the 256 8-bit patterns should appear in the sequence equally often. Additionally, the byte sequence should be as uncorrelated as possible. This means, for example, that for any two given bytes, the probability of their appearing together should be no greater than what is dictated by their appearance as individual bytes.

- The pseudorandom byte sequence is a function of the encryption key. To foil brute-force attacks, the encryption key should be as long as possible, subject to, of course, all the other practical constraints. A desirable key length these days is 128 bits.

- With a properly designed pseudorandom byte generator, a stream cipher for a given key length can be as secure as a block cipher using keys of the same length.

- The next section presents pseudorandom byte generation for the RC4 stream cipher. (Lecture 10 will go into the subject of pseudorandom number generation for general cryptographic applications.)

- As you would expect, a stream cipher is particularly appropriate for audio and video streaming. A stream cipher is also frequently used for browser – web-server links. A block cipher, on the other hand, is more appropriate for file transfer, etc.

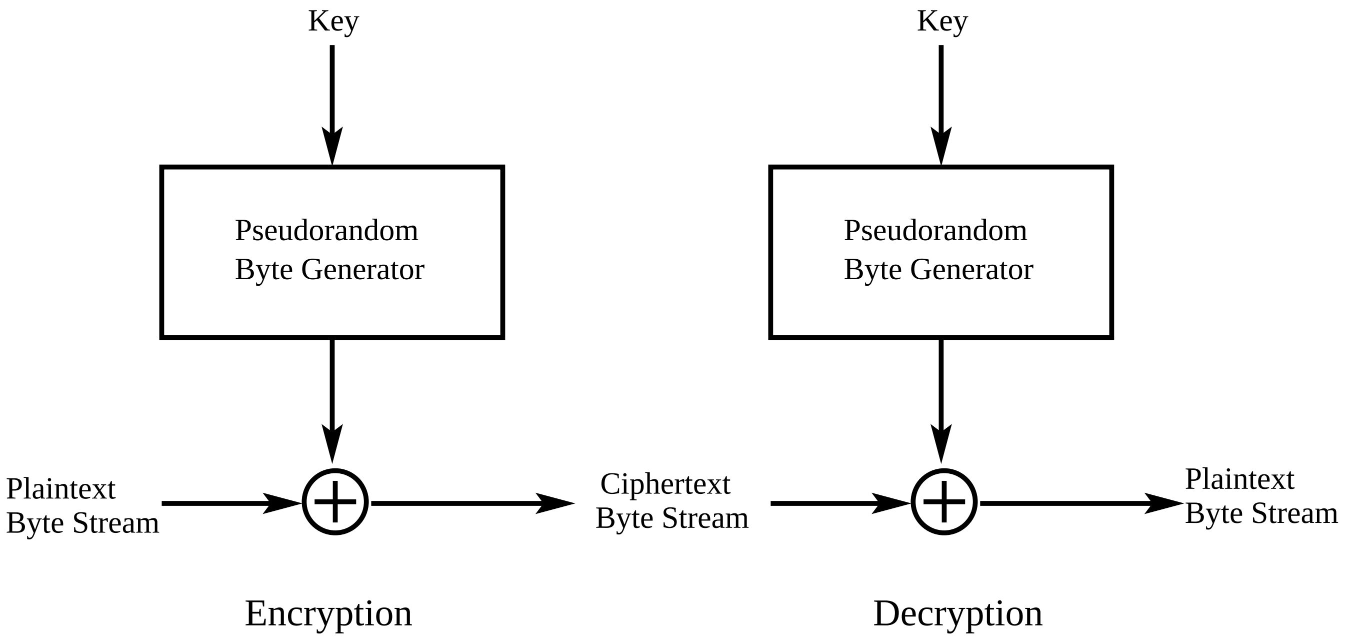


Figure 8: *Operation of a stream cipher.* (*This figure is from Lecture 9 of "Computer and Network Security" by Avi Kak*)

9.7: THE RC4 STREAM CIPHER ALGORITHM

- As mentioned earlier in Section 9.6, a key component of a stream cipher is the pseudorandom byte sequence generator.

- We will now go through the pseudorandom byte sequence generator in the RC4 algorithm.

- RC4 is a variable key length stream cipher with byte-oriented operations.

- Fundamental to the RC4 algorithm is a 256 element array of 8-bit integers. It is called the **state vector** and denoted S.

- The state vector is initialized with the encryption key. The exact initialization steps are as follows:

  – The state vector S is initialized with entries from 0 to 255 in the ascending order. That is

$$\begin{array}{rclcl} S[0] & = & 0x00 & = & 0 \\ S[1] & = & 0x01 & = & 1 \\ S[2] & = & 0x02 & = & 2 \\ S[3] & = & 0x03 & = & 3 \\ ..... & & & & \\ ..... & & & & \\ S[255] & = & 0xFF & = & 255 \end{array}$$

– The state vector S is further initialized with the help of another temporary 256-element vector denoted T. This vector also holds 256 integers. The vector T is initialized as follows

  * Let's denote the encryption key by the vector K of 8-bit integers. Suppose we have a 128-bit key. Then K will consist of 16 non-negative integers whose values will be between 0 and 255.

  * We now initialize the 256-element vector T by placing in it as many repetitions of the key as necessary until T is full. Formally,

$$T[i] \quad = \quad K[i \mod keylen] \qquad for\ 0 \leq i \leq 255$$

  where *keylen* is the number of bytes in the encryption key.

In other words, *keylen* is the size of the key vector K when viewed as a sequence of non-negative 8-bit integers.

– Now we use the 256-element vector T to produce the **initial permutation** of S. This permutation is according to the following formula that first calculates an index denoted j and then swaps the values $S[i]$ and $S[j]$:

```
j  =  0
for i = 0 to 255
    j  =  ( j + S[i] + T[i] )  mod  256
    SWAP  S[i], S[j]
```

This algorithm is generally known as the **Key Scheduling Algorithm** (KSA).

– There is no further use for the temporary vector T after the state vector S is initialized as described above.

– Note that the encryption key is used only for the initialization of the state vector S. It has no further use in the operation of the stream cipher.

– Note also that initialization procedure for the state S is just a permutation of the integers from 0 through 255. Each integer in this range will be in one of the elements of S after initialization. This happens because all that the initialization does is to swap the elements of S according to the secret key.

- Now that the state vector S is initialized, we are ready to describe how the **pseudorandom byte stream** is generated from the state vector. Recall that when you are using a stream cipher, as each byte of the plaintext becomes available, you XOR it with a byte of the pseudorandom byte stream. The output byte is what is transmitted to the destination.

- The following procedure generates the pseudorandom byte stream from the state vector

```
i, j  =  0
while ( true )
    i  =  ( i + 1 )  mod  256
    j  =  ( j + S[i] )  mod  256
    SWAP S[i], S[j]
    k  =  ( S[i]  +  S[j] )  mod  256
    output S[k]
```

  Note how the state vector S changes continuously by the swapping action at each pass through the **`while`** loop. In other words, the state of the pseudorandom number generator changes dynamically as the the numbers are being generated.

- The above procedure spits out $S[k]$ for the pseudorandom byte stream. The plaintext byte is XORed with this byte to produce an encrypted byte.

- The pseudorandom sequence of bytes generated by the above algorithm is also known as the **keystream**.

- Theoretical analysis shows that for a 128 bit key length, the period of the pseudorandom sequence of bytes is likely to be greater than 10^{100}.

- Because all operations are at the byte level, the cipher possesses fast software implementation. For that reason, RC4 remains the mostly widely used software stream cipher.

- RC4 is used in the **SSL/TLS** (Secure Socket Layer / Transport Layer Security) standard for secure communications between web browsers and web servers.

- RC4 is also used in the **WEP** (Wired Equivalent Privacy) protocol and the newer WiFi Protected Access (**WPA**) protocol that are part of the IEEE 802.11 wireless LAN standard.

- We will next focus briefly on some specific weaknesses of RC4 as it is used in WEP for wireless security in small networks. To understand these weaknesses, you must first understand how RC4 is used in wireless network communications.

9.8: WEP FOR WiFi

- WiFi is a popular name for WLAN (Wireless Local Area Network). With WiFi, computers connect wirelessly to the internet through an *Access Point* (AP). A single AP, also referred to as a *hotspot*, typically has a range of around 30 meters indoors. Wider coverage (such as campus wide coverage) can be achieved by using multiple APs that are connected through a wired distribution system. All the APs working together in this manner constitute a LAN (local area network) that in internet terms constitutes a *subnet*. It is a subnet because, logically speaking, it is *bounded* by a single router. A network indentifier, called as SSID (Service Set Identifer) is associated with each WLAN. SSID is also known as a network name. At Purdue, you now have two networks, PAL2 and PAL3, operating simultaneously. [Your home WiFi, likely to be driven by a LinkSys, NetGear, D-Link, etc., router, constitutes a LAN in which the router doles out Class-C addresses in the 192.168.0.0 – 192.168.255.255 range. The campus-wide WiFi at Purdue also constitutes a LAN that uses Class-A addresses in the 10.0.0.0 – 10.255.255.255 range. Note that Class-C networks typically use a 24-bit *subnet mask* — which is the number of leading bits reserved for network addressing. That leaves only 8-bits for host addressing, which makes for a maximum of 254 hosts (since one address must be reserved for the router itself and one is used as a broadcast address by the router) that you can have in such a network. Finally, in the context of SOHO (Small Office and Home) WiFi, "router" and "AP" are used interchangeably. For a campus-wide WiFi, on the other hand, you'll obviously have multiple APs for a single logical router.

Note that it is the router's job to assign an IP address to a connecting host and to serve as a gateway to the internet.]

- WiFi communications are based on a set of standards commonly referred to as the IEEE 802.11 standards. WiFi uses a set of bands, consisting of 20 MHz channels, at the 2.4 GHz and 5 GHz frequencies.

- WEP is a protocol for encrypting the packets that are transmitted over a wireless communication link according to the IEEE 802.11 standards. Despite its shortcomints, it is still the most commonly used security protocol for home and small business wireless networks.

- The WEP protocol requires each packet to be encrypted separately with its own RC4 key. So if a TCP packet contains, say, a payload of 1024 bytes, those bytes would be encrypted by RC4 using a key specific to that packet. [The TCP protocol is discussed in Lecture 16.]

- **As made clear by the next bullet, there is a very important reason for why no two packets should be encrypted with the same RC4 key.**

- If the same keystream S is used for two different plaintext byte streams P_1 and P_2, an XOR of the corresponding ciphertext streams becomes independent of the keystream because

$$C_1 \oplus C_2 \quad = \quad (P_1 \oplus S) \oplus (P_2 \oplus S) \quad = \quad P_1 \oplus P_2$$

This can create a backdoor to extracting the plaintext stream from the ciphertext stream. All you have to do is to XOR the ciphertext in each packet with the ciphertext stream in a packet in which a reasonably large number of bytes are set to 0.

- The RC4 key for each packet is a simple concatenation of a 24-bit Initialization Vector (IV) and the root key, which, at least in the context of home wireless networks, is sometimes referred to as the AP's *security code.* The root key corresponds to the pass phrase you enter in your home wireless router. While the root key remains fixed over all the packets, you increment the value of IV from one packet to the next. [The most commonly used WEP encryption is based on 40-bit root keys, although many AP vendors also support 104-bit WEP encryption. The official WEP standard only calls for 40-bits for the root key. The root key for my home wireless AP consists of 10 hex characters, meaning it is a 40-bit key. While we are on the subject of root keys, note that there are AP vendors who advertise 128-bit WEP encryption. It is a misleading claim that is meant to create the impression that their APs support superior encryption — their 128 bits are merely a sum of a 24-bit IV that all APs must use for WEP and a 104-bit root key.]

- WEP then computes the CRC32 checksum of the data to be encrypted in the packet. [CRC stands for Cyclic Redundancy Check. It is a generalization of the commonly used parity check that is used to guard against data corruption during transmission. CRC32 gives us a 32-bit checksum. Think of it as a 32-bit digital signature. In WEP, this CRC32 signature is called **Integrity Check Value** (ICV). Finding CRC32 of a binary data stream amounts to dividing the data bit pattern (which could be the bits in an entire file) by an irreducible (or sometimes reducible) polynomial of degree 32. This would obviously leave a residue polynomial whose highest degree would only be 31. The bit pattern corresponding to the residue would therefore only be 32 bits long. Note also CRC1, which is the same thing as using a parity bit for error detection, amounts to using $x + 1$ as the divisor polynomial. While it is possible (and fairly common) to use reducible polynomials, their error detection capabilities are less effective.]

- The RC4 key for a packet is then used to encrypt the data followed by its ICV value mentioned in the small-font note above.

- The biggest problem with WEP in a typical usage scenario is that the root key remains fixed for long periods of time (in home use, people almost never change their root keys) and the IV has only 24 bits in it. **This implies that distinct keystreams can be generated for only 2^{24} (around 16 millions) different packets.** This implies that the same keystream will be used for different packets in a long session. How frequently that can happen depends on how the IVs are generated.

- As mentioned earlier, the 3-byte IV is prepended to the root key. Since the IV is sent in plaintext, anyone with a packet sniffer can directly see the first three bytes of the RC4 key used for a packet. An 802.11 frame that is encrypted with WEP looks like:

```
 ------------------------------------------------------          ---
| 802.11 Header                                        |            |
 --------------------------------------------------------           | Plaintext
| BSSID  |  Initialization Vector (IV)  |  Dest. Address |          |
 --------------------------------------------------------        ---
|                  Logical Link Control                  |          |
 --------------------------------------------------------           |
|               Subnetwork Access Protocol Header        |          |
 --------------------------------------------------------           | RC4
|                         Data                           |          | Encrypted
 --------------------------------------------------------           |
|                  Integrity Value Check                 |          |
 --------------------------------------------------------           |
                                                                 ---
```

- Note that WPA also uses RC4. WPA provides enhanced security because it uses a 48-bit Initialization Vector. Additionally, WPA hashes the Initialization Vector before combining it with the root key. As to what it means to hash something, that is the subject of Lecture 15.

9.8.1: The Klein Attack for Figuring Out the WEP Root Key

- This attack is based on Andreas Klein's combinatorial analysis of the pseudorandom sequence produced by the RC4 algorithm. A complete citation to the paper is: Andreas Klein, "Attacks on the RC4 Stream Cipher," *Designs, Codes, and Cryptography*, Vol. 48(3), pp. 269-286 2008.

- Before we review the basic notions that go into the Klein attack, we will first write more compactly the RC4 key scheduling algorithm and the pseudorandom byte generation algorithm that were explained in Section 9.7. The more compact versions of these two algorithms are shown as Algorithm 1 below and Algorithm 2 on the next page.

Algorithm 1 Algorithm 1: RC4 Key Scheduling

```
 1: {initialization}
 2: for $i = 0$ to $n - 1$ do
 3:         $S[i] \leftarrow i$
 4: end for
 5: $j \leftarrow 0$
 6: {generate a random permutation}
 7: for $i$ from 0 to $n - 1$ do
 8:         $j \leftarrow (j + S[i] + K[i \ mod \ length(K)]) \ mod \ n$
 9:         Swap $S[i]$ and $S[j]$
10: end for
```

Algorithm 2 Algorithm 1: RC4 Pseudorandom Generator

```
{initialization}
$i \leftarrow 0$
$j \leftarrow 0$
{generate pseudorandom sequence}
loop
    $i \leftarrow (i+1)\ mod\ n$
    $j \leftarrow (j + S[i])\ \ mod\ \ n$
    Swap $S[i]$ and $S[j]$
    $k \leftarrow (S[i]\ +\ S[j])\ \ mod\ \ n$
    output $S[k]$
end loop
```

- Klein has shown that strong correlations exist in the byte sequence produced by the pseudorandom byte generation algorithm. These correlations are expressed in the form of probabilities of the output pseudorandom sequence satisfying certain constraints vis-a-vis the the values of the state vector S.

- The attack proposed by Klein is a plaintext-ciphertext attack. For the case of WEP, an easy way to collect the needed plaintext-ciphertext pairs is for the attacker's wireless interface to send repeated ARP requests to the wireless AP being attacked. Each transmitted ARP request will elicit a reply whose 802.11 frames will follow the format shown in the previous section. Even though the attacker will only see the ciphertext for the encrypted portion of these 802.11 frames, he/she can make good guesses for the fields that come before the "Data" field. For example, the information that is placed in the SNAP header field (shown as "Subnetwork Access Protocol Header" in the figure in the previous section) would be guessable by the attacker. For example, the first three bytes of SNAP are generally the same as the first three bytes of the AP's MAC address. [The Klein attack and its successor the PTW attack presented in the next subsection use the ARP packets to collect plaintext-ciphertext pairs. (We will have more to say about ARP in Section 23.3 of Lecture 23.) Suffice it to say here that ARP stands for Address Resolution Protocol. It is used by the machines in a LAN to figure out the physical-layer MAC addresses for the other machines in the LAN. For example, if your laptop hooked to a wireless LAN needs to figure out how to send a packet to another laptop in the LAN whose IP address happens to be 192.168.1.105, your laptop will broadcast on the LAN an ARP packet asking the 192.168.1.105 machine to respond with its MAC address. The first 15 bytes of an ARP packet are transmitted in plaintext form even when the data payload is encrypted. You should also know that ordinarily there may not be a sufficient number of ARP packets available for mounting a meaningful attack. So a part of the attack strategy is to have a large number of ARP requests going out from an attacking machine so that a sufficiently large number of response packets can be harvested for the analysis you are going to read about in what follows.] **These plaintext bytes can be XOR'ed with the ciphertext bytes to recover several initial bytes of the pseudorandom sequence that was generated by the RC4 algorithm.** So, henceforth, we will assume that our goal is to figure out the bytes of the root key from the available bytes of the pseudorandom sequence.

- There are two main theoretical results derived by Klein that play a critical role in the attack. The first of these is

$$Prob\,(S[j] + S[k] \;\equiv\; i \; mod \; n) \quad = \quad \frac{2}{n} \qquad (1)$$

where n is the modulus integer 256 used in RC4. In order to understand what this formula is telling us, you have to pay close attention to the notation whose meaning is derived from the description in Algorithm 2. The variable i above is as set in Algorithm 2. On account of line 6 of Algorithm 2, the value of i for the first output byte will be 1, for the second output byte 2, and so on. Obviously, we can refer to i as an observable variable since we can infer its value for each output byte. The entity $S[k]$ is the byte that is output in line 10 of the algorithm. Assume that we can see the pseudorandom byte stream produced by Algorithm 2. Obviously, then, $S[k]$ would also be observable. On the other hand, the entity $S[j]$ is the value of the state vector at index j that was used in the calculation of $S[k]$ for a given value for i. So, as far as someone observing the pseudorandom byte sequence is concerned, $S[j]$ is internal to the byte generator. The above formula tells us that for an i for a given output byte, the probability of the output byte plus the state vector byte $S[j]$ being equal to $i \ mod \ n$ is $2/n$.

- Therefore, for the first output pseudorandom byte, we can say that $Prob(S[j] + S[k]) \ = \ 1$ is $2/256$ where $S[k]$ is the value of the byte that is output and $S[j]$ state vector byte that goes into the calculation of the output byte.

- That brings us to the second main theoretical result of Klein: For a given i that indexes an output byte according to line 6 of Algorithm 2, let's now consider all $c \in \{0, \ldots, n-1\}$ but with $c \neq i$, we have

$$Prob\,(S[j] + S[k] \;\equiv\; c \; mod \; n) \quad = \quad \frac{n-2}{n(n-1)} \qquad (2)$$

where the other symbols are to be interpreted as earlier.

- The basic form of the attack consists of assuming that you already know $K[0]$ [For WEP, you know the first three bytes of the key used for each packet since those are the three bytes of the Initialization Vector that is transmitted in plaintext. Klein's discussion is based on the premise that initially you know nothing about the key K and you start by assuming a value for the first byte of the key. That is because Klein's paper is about attacking RC4 in general. To apply the Klein attack to WEP, you start with knowing the first three bytes of the key and then using Klein's recursive reasoning to figure out the bytes of the root key.] Klein then shows you can guess a value for $K[1]$ that will be the correct value with a high probability. This is followed by recursively guessing the values for the rest of the key bytes. [The discussion that follows is for what Klein refers to as the 1-round attack. A round for the RC4 algorithm refers to the production of n output bytes with $n = 256$. That is, the attack will be based solely on the first 256 bytes produced by the pseudorandom byte generator.]

- The reasoning for making a good guess for $K[1]$ goes as follows (*these are reproduced from the paper by Klein that was cited at the beginning of this section*):

  – We start by examining the first two bytes produced by the Key Scheduling Algorithm (Algorithm 1). For the first iteration,

$i = 0$ and the value of j takes the value $K[0]$ line 8, which is followed by swapping $S[0]$ with $S[K[0]]$. In the second iteration, $i = 1$, j is now increased by $S[1] + K[1]$, and entry of $S[j]$ is moved to $S[1]$.

– Since we start with $S[j] = j$ for all j, we can show that, at the end of the second iteration of the loop in lines 5 through 11 of Algorithm 2, the value of the output byte $S[1]$ is $t = K[0] + K[1] + 1$ in all except for those listed below:

1. If it should happen that $K[0] = K[1]$, then the value of the second output byte is $t = 0$.

2. If it should happen that $K[0] = 1$ and $K[1]$ is neither equal to 0, nor to $n-1$, then one can show that $t = K[0]+K[1]$.

3. If it should happen that $K[0] \neq 1$ and $K[1] = n - 1$, in this case $t = 0$.

4. If it should happen that $K[0] \neq 1$ and $K[0]+K[1] = n-1$, in this case $t = K[1]$.

The important conclusion here is that the value t of the second output byte $S[1]$ is an easily computable function of $K[0]$ and $K[1]$.

- All of the reasoning presented above applies up to the moment the second byte is output by Algorithm 2. In the remaining iterations of the algorithm, what is stored in $S[1]$ will only change when the value j becomes 1. Klein has shown that when the key length equals n and when the key bytes are independent, the probability that $S[1]$ will change during the production of the first 256 bytes is $(1 - 1/n)^{n-2} \approx 1/e$.

- The reasoning presented so far has told us how the value t of the second output byte from pseudorandom generator is related to the key bytes $K[0]$ and $K[1]$. Our next goal is to guesstimate t from the first two bytes of the pseudorandom sequence. Obviously, if we can make a correct guess for t, we can then find $K[1]$ since we know how t depends on $K[0]$ and $K[1]$.

- With regard to guessing the value of t, let's assume that the attacker has a large number of first rounds of different runs of pseudorandom sequence available to him/her. In WEP, these may corresponding to the different values for the 3-byte Initialization Vector (IV) we talked about in Section 9.8. We know from Equation (1) that in each of these sequences, the following must be true for the first byte $S[j] \equiv 1 - S[k]$ with a probability of $2/n$. Klein used the correlations in the output pseudorandom sequence, expressed by the equations (1) and (2) shown earlier, to establish the following result:

$$\begin{aligned} Prob\,(t \equiv (1 - S[k])\ mod\ n) &\approx \frac{1}{e}\cdot\frac{2}{n} + (1-\frac{1}{e})\cdot\frac{n-2}{n(n-1)} \\ &\approx \frac{1.36}{n} \end{aligned} \tag{3}$$

Pay close attention to what the left hand side is saying: It is asking for the probability of t as defined previously being $K[0] + K[1]+1$ being equal to $1-S[k]$, something we can calculate from the observables. The right hand side tells us that this probability is $1.36/n$.

- On the strength of the above probability, the attacker now does the following (quoting Klein): "For a number of initialization vectors, the attacker observes the **first** byte x_i of the pseudo-random byte generator and, for each value first-byte value, the attacker calculates $t_i = 1 - x_i$. (The index i here is to the i^{th} pseudorandom sequence examined.) The fraction of the t_i that have the correct value of t (meaning the value $K[0] + K[1] + 1$) is about $1.36/n$. All other possible values for t_i will have a relative frequency of $1/n$. If the number of pseudorandom sequences examined is large enough, we can be sure that the most frequent value is the correct value."

- We thus have a procedure for calculating the byte $K[1]$ of the key assuming that we have a good guess for the first byte $K[0]$. We are able to guess the correct key byte for $K[1]$ with a probability of $1.36/256$, which is higher than the probability of $1/256$ for what a monkey would guess for $K[1]$.

- Klein has shown how the same rationale can be extended to estimate $K[2]$ and the rest of the key bytes. The only drawback to the procedure being that the calculation of the key byte $K[i]$ depends on all the previous key bytes $K[0]$, $K[1]$, ... , $K[i-1]$.

9.8.2: The PTW Attack for Figuring Out the WEP Root Key

- Basically, this attack is founded on the same theoretical principles as the Klein attack. Therefore, it is important to understand the Klein attack in order to understand the PTW attack.

- The acronym PTW stands for the authors Erik Tews, Ralf-Philipp Weinmann, and Andrei Pyshkin. The attack is described in their publication "Breaking 104 Bit WEP in Less Than 60 Seconds," (in *Lecture Notes in Computer Science*, pp. 188-202, Springer, 2007). Most of the programs that are popular today for breaking WEP are based on this work.

- The PTW attack removed an important shortcoming of the Klein attack's need to calculate the key bytes recursively. So if an error was made in the calculation of one of the bytes, the rest of the key bytes would be wrong also. In the PTW attack, the key bytes are calculated independently.

- PTW's attack is based on their demonstration that if we know the first i key bytes (which we always do in WEP with $i = 3$), we can guess the sum of the key bytes indexed from i to $i + k$ with a probability of 1.24/256. The PTW algorithm constructs a guess for this summation for every key byte from $i = 3$ to $i = 16$. The actual key bytes are then calculated from the guesses for the sums.

- Although it is incredibly fast and requires not much data, the main limitation of PTW is that it can only crack 40 and 104 bit keys.

- The last Homework problem at the end of this lecture is for you to use the `aircrack-ng` package to try to break WEP in a wireless network.

- Before ending our presentation of the security issues related to WEP, we should mention another attack, known as the FMS attack, that was the main form of cracking WEP before the Klein attack and its successor, the PTW attack, came along. The FMS attack, named after Scott Fluhrer, Itsik Mantin, and Adi Shamir, is presented in their publication "Weaknesses in Key Scheduling Algorithm of RC4," *Lecture Notes in Computer Science*, pp. 1-24, 2001. With the FMS attack, it is possible to guess the key bytes when the 3-byte Initialization Vector satisfies certain properties. However, the attack require a large amount of data, of the order of 4 million packets. In 2004, this attack was made stronger by someone using the pseudonym KoreK. With the KoreK attack, the key bytes could be guessed with about 500,000 packets.

9.8.3: Using the `aircrack-ng` Package to Break WEP in Under a Minute

- Download the **`aircrack-ng`** package with your Synaptic package manager into your Ubuntu laptop. You can use this package to mount the super-fast PTW attack to crack the encryption key being used in a locked WiFi.

- The WEP security exploit described in this section works only if the version number of your Ubuntu install is 13.10 or higher. With the older versions of Ubuntu, you had to make certain changes to the wireless driver supported by the Linux kernel so that your wireless interface could establish fake associations and fake authentications with the attacked access point. Using these fake associations, your wireless interface would mount a replay attack on the attacked access point for the purpose of acquiring a large number of ARP packets with different initialization vectors. [ARP stands for Address Resolution Protocol. We described ARP rather briefly in Section 9.8.1. Further information regarding ARP can be found in Section 23.3 of Lecture 23.] With the more recent versions of Ubuntu, you need to change nothing at all in your machine in order to mount an attack on the WEP security of a wireless access point. [This comment in red applies to mounting WEP exploits with machines that have older versions of Ubuntu installed: In general, any tinkering with any part of the kernel of an OS would require that the entire kernel be recompiled. A good thing about Linux is that the part of the kernel that deals with the operation of the wireless and bluetooth forms a separate code tree that can be changed and compiled separately from the rest of the OS. This code tree used to be called `compat-wireless` and was made available by the Linux wireless developers community at `http://linuxwireless.org`. Recently, the `compat-wireless` package was renamed `backports`. Visit the wiki at `https://backports.wiki.kernel.org` for the `backports` package.]

- Before you can attack a wireless Access Point for its WEP security, you'd need to identify it with its MAC address and the channel it is using. This you can do by running a command like '`iwlist wlan0 scan`' that shows all the APs that are within the radio range of your laptop and then choosing the one you are going to attack. (You'll hopefully choose an AP that is your own.) Record the MAC address of the AP from the output of the `iwlist` command and also note the channel number. We will denote this MAC address by `xx:xx:xx:xx:xx:xx` and the channel number by `yy`.

- The WEP exploit requires that you create what is known as a Monitor Mode of your wireless interface and you would want this mode to run with its own MAC address. Normally, your wireless interface operates in what is known as the Managed Mode. The idea is to have the two modes operating concurrently on the same physical wireless device in a computer. [In general, the hardware in your laptop for wireless communications (which is also referred to as an 802.11 wireless card after the famous IEEE 802.11 protocol for the operation of wireless devices in computer networks) can support *wireless intefaces* that can be operated in one or more of the following six modes: **(1) Master Mode** – A wireless interface in the Master Mode is often referred to as an Access Point (AP) or a Base Station. **(2) Managed Mode** — This is the normal mode of using your 802.11 card in your laptop. In this case, your wireless interface associates with a single AP serving as a central hub for all traffic emanating from your laptop or intended for it. A wireless interface in this mode will reject all incoming packets coming off the AP but not intended for it. The wireless interface will also reject all packets coming off any other 802.11 devices within the radio range. This mode is also known as the Infrastructure Mode. A wireless interface operating in the Managed Mode is also referred to as a "802.11 station." **(3) Monitor Mode** — This allows the wireless interface to capture packets going to and coming off an AP without having to associate with it. The Monitor Mode in the context of wireless is analogous to the promiscuous mode for an ethernet interface for wired LANs. In the Monitor Mode, a wireless interface will also be able to capture the ARP packets that the attacked AP may be broadcasting to the other 802.11 stations in the same channel. **(4) Ad-Hoc Mode** — In this mode the different 802.11 wireless interfaces can talk to one another directly without having to go through an AP. **(5) Mesh Mode** — In this mode, two 802.11 devices can communicate with each other if they have at least one other such device in the intersection of their radio ranges. And, finally, **(6) Repeater Mode** — A wireless interface operating in this mode merely re-broadcasts the packets it receives. This mode is used to extend the range of an AP.] [If you need to know what modes the 802.11 wireless card in your laptop can operate in and what encryption algorithms it can call upon, you have to first find out what name the Physical Layer of the OSI representation of the TCP/IP protocol is using for your wireless card. This you can do by executing the command '`airmon-ng`' without options. The information returned by this command on my laptop tells me that the Physical Layer name of my wireless card in `phy0`. Subsequently, you can see a large number of attributes of the `phy0` object by running the command '`iw phy phy0 info`'. You can see the modes supported by your wireless card by executing '`iw phy phy0 info | grep -A8 modes`' where the option `-A8`' tells `grep` to show eight additional lines beyond each matching line. Running this command tells me that the 802.11 card in my laptop can support just the Managed and the Monitor modes.]

- You create a wireless interface in Monitor Mode by executing a command like

```
airmon-ng start wlan0
```

where `wlan0` is the name of the wireless interface in its normal mode — meaning the Managed Mode. The command shown above will create new Monitor-Mode wireless interface named `mon0`.

- A wireless interface created in the Monitor Mode needs a MAC address that's distinct and different from that of the Managed Mode wireless interface. This you can do by executing the `macchanger` command as follows: [For this, you would need to first install the `macchanger` module through your Synaptic package manager or `apt-get`.]

```
    macchanger -m 00:11:22:33:44:55 mon0
```

  which assigns the MAC address `00:11:22:33:44:55` to the Monitor-Mode interface `mon0`. Since you are free to conjure up any reasonable looking MAC address for this, you might as well choose something that is relatively easy to type and remember. Since the new interface created by '`airmon-ng start wlan0`' will be on, you'd first need to shut it down with a command like '`ifconfig mon0 down`' before assigning it a new MAC address.

- What you see next is a convenience script in which are packaged the various steps listed above for setting up the Monitor-Mode wireless interface and assigning it a MAC address. The script makes it easier to mount the exploit multiple times. [Your initial attempts may not succeed for several reasons. In particular, an attempt may fail if the number of ARP packets you captured did not yield a sufficiently large haul of IVs (Initialization Vector in the RC4 algorithm). How many IVs can be harvested from a given collection of ARP packets depends on how busy the wireless LAN is.] The script shown below first removes the dump file created in the previous run of the exploit. It then tries to find out if you had created the `mon0` interface previously. This is accomplished by examining the output produced by the `ifconfig` command for the presence of the `mon0` string. If a previously constructed instance of `mon0` is detected, the script invokes the command '`airmon-ng stop mon0`' to kill it. Finally, the `airodump-ng` command you see in the script tells `aircrack-ng` that it should start collecting the packets whose transmission you will soon initiate.

```
#!/bin/sh

#  StartMonitorModeInterface.sh
#
#  by Avi Kak (kak@purdue.edu)

#  Run this is a separate window and wait for the last command shown
#  to kick in to start collecting the packets and dumping them in the file
#  specified with the '-w' option.

#  After you have collected sufficient packets, kill the script
#  with ctrl-C.

#  Note that yy is the channel number and xx:xx:xx:xx:xx:xx is the MAC
#  address of the Access Point you want to attack.

rm -f mydumpfile* replay_arp*
sleep 5
ifconfigOut=`ifconfig mon0 2>&1`
cleanedup="$(echo $ifconfigOut | tr -d ' ')"
if [ `expr $cleanedup : '.*errorfetching.*'` -eq 0 ]
then
    echo killing old Monitor-Mode interface mon0
    airmon-ng stop mon0
fi
sleep 5
echo starting new Monitor-Mode interface mon0
```

```
airmon-ng start wlan0
sleep 5
ifconfig mon0 down
sleep 5
macchanger --mac 00:11:22:33:44:55 mon0
sleep 5
ifconfig mon0 up
sleep 5
airodump-ng -c yy -w mydumpfile --bssid xx:xx:xx:xx:xx:xx mon0
```

- With the help of the script shown above, attacking the WEP security of an AP consists of just the following three steps:

Step 1: As root, execute the shell script `StartMonitorModeInterface.sh`. However, before you execute the script, you must change `xx:xx:xx:xx:xx:xx` in the script to the MAC address of the Access Point you are attacking and `yy` to the channel number that your laptop is using to communicate with the AP.

Step 2: In a separate window, execute the following command as root in order to inject and replay the ARP packets from your laptop to the AP:

```
aireplay-ng -2 -p 6000 -c FF:FF:FF:FF:FF:FF -b xx:xx:xx:xx:xx:xx -h 00:11:22:33:44:55 mon0
```

You'd obviously need to replace `xx:xx:xx:xx:xx:xx` by the MAC address of the AP you are attacking. In the command line shown above, the option '`-2`' specifies the *attack mode*. In this mode, before the attack is lauched, you are shown the ARP packet that will be used for injected and replay. If you enter 'no' to the packet being shown to you, you'll be shown another packet, and so on, until you accept one. [The manpage for `aireplay-ng` says that there are nine different attack modes. With the older versions of Ubuntu, we used to use the option '`-3`'. In this option, `aircrack-ng` listens for an ARP packet and then retransmits it back to the AP. That, in turn, causes the AP to broadcast the ARP packet again with a different Initialization Vector (IV).] The '`-b`' option stands for BSSID, which is the MAC of the AP you are attacking. The option '`-h`' is the source MAC, that is, the address of the Monitor-Mode wireless interface on your laptop. In case you are wonderfing about the '`-p`' option, it sets the "frame control word" in hex — according to the manpage for the `aireplay-ng` command. The '`-c`' option specifies what constraints would apply to the destination MAC in the packets that will be captured by the `mon0` interface. Since `mon0` is meant to be running in the promiscuous mode, by supplying the value shown, we make it possible for `mon0` to capture packets that may actually be meant for other nodes in the local wireless LAN.

Step 3: Both the windows, the window in which you are running the `StartMonitorModeInterface.sh` script and the window in which you are running the command in the previous step, will show a continuously changing readout, the first for the packets that are being dumped in the designated dump file and the second for the ARP packets that are being captured. After you have captured a large enough collection of packets (say, around 100,000 packets), it's time to kill both of those jobs. Now execute as root the following command line to crack the WEP:

```
aircrack-ng -b  xx:xx:xx:xx:xx:xx  mydumpfile-01.cap
```

where, again, `xx:xx:xx:xx:xx:xx` is the MAC address of the access point that you attacked. If your attack was successful, it will very quickly display the WEP key being used by the access point. If your exploit was successful, the above command will show you the WEP key. If not, you will be asked to repeat the exploit.

9.9: HOMEWORK PROBLEMS

1. A block cipher algorithm in its basic form is almost never used for encrypting long messages. Why? How are block ciphers deployed in practice if you want to encrypt long messages?

2. Even with the chaining modes described in this lecture, one of the difficulties with using a block cipher — even the strongest block cipher — is the problem of padding. Since it is unlikely that the length of an arbitrary message or a file would be an exact multiple of the block size used in the block cipher, the two end points of a secure communication link must have in place some sort of a protocol regarding how to pad the plaintext so that its overall length is an exact multiple of the block size. **With a stream cipher, such as RC4, we do not have to face this problem. That, along with the fact that RC4 possesses a very efficient software implementation, made RC4 the cipher of choice in the SSL/TLS protocols used for secure transfer of documents between web servers and web browsers.** However, it's good to keep in mind the fact that, despite (and especially because of) its popularity, RC4 does possess some security vulnerabilities that are caused by the correlations between the first few bytes of the keystream and the

corresponding bytes of the state vector (as initialized by the encryption key). Check out the Wikipedia page on RC4, along with the references listed therein, and describe in greater detail these security vulnerabilities.

3. What are the essential elements of the RC4 algorithm? What networking applications use the RC4 stream cipher?

4. What might be the main reason for why the keystream generation in RC4 has a very efficient software implementation?

5. What is the problem with WEP? What makes it an "unsafe" protocol for wireless networking?

6. What makes WPA a more secure protocol?

7. **Programming Assignment:**

   Write a Perl or Python script that implements RC4 for encryption and decryption. Your script should read a sound file in the wave format and produce an encrypted version of the same file. Try listening to both the original and its encrypted version through a sound player on your computer. Now use your decryption program to recover the original from the encrypted version. Verify the correctness of decryption by listening to the sounds again.

Your key length should be 16 ASCII characters that you enter from the keyboard. (These would translate into a 128 bit encryption key.) In addition to testing your scripts on your own sound files, you may also wish to use the wave files available on the course web site. If using Python, use the `wave` module to read and write wave format files. If using Perl, use the `Audio::Wav` module from `www.cpan.org`.

8. **Programming Assignment:**

   Using the three shell scripts in Section 9.8.3, mount a super-fast PTW attack to crack the WEP encryption key being used at your home WiFi. See how your success rate depends on the distance of your laptop from the access point you are attacking, on the rate you inject ARP packets into the channel, etc. How would you modify the `inject_arp.sh` script for attacking WPA/WPA2 access points?