Lecture 2: Classical Encryption Techniques

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 19, 2015
10:58am

Goals:

- To introduce the rudiments of encryption/decryption vocabulary.
- To trace the history of some early approaches to cryptography and to show through this history a common failing of humans to get carried away by the technological and scientific hubris of the moment.
- Python scripts that give you pretty good security for confidential communications. Only good for fun, though.

CONTENTS

2.1: BASIC VOCABULARY OF ENCRYPTION AND DECRYPTION

plaintext: This is what you want to encrypt

ciphertext: The encrypted output

enciphering or encryption: The process by which plaintext is converted into ciphertext

encryption algorithm: The sequence of data processing steps that go into transforming plaintext into ciphertext. Various parameters used by an encryption algorithm are derived from a secret key. In cryptography for commercial and other civilian applications, the encryption and decryption algorithms are made public.

secret key: A secret key is used to set some or all of the various parameters used by the encryption algorithm. **The important thing to note is that, in classical cryptography, the same secret key is used for encryption and decryption.** It is for this reason that classical cryptography is

also referred to as symmetric key cryptography. **On the other hand, in the more modern cryptographic algorithms, the encryption and decryption keys are not only different, but also one of them is placed in the public domain.** Such algorithms are commonly referred to as asymmetric key cryptography, public key cryptography, etc.

deciphering or decryption: Recovering plaintext from ciphertext

decryption algorithm: The sequence of data processing steps that go into transforming ciphertext back into plaintext. In classical cryptography, the various parameters used by a decryption algorithm are derived from the same secret key that was used in the encryption algorithm.

cryptography: The many schemes available today for encryption and decryption

cryptographic system: Any single scheme for encryption and decryption

cipher: A cipher means the same thing as a "cryptographic system"

block cipher: A block cipher processes a block of input data at a time and produces a ciphertext block of the same size.

stream cipher: A stream cipher encrypts data on the fly, usually one byte at at time.

cryptanalysis: Means "breaking the code". Cryptanalysis relies on a knowledge of the encryption algorithm (that for civilian applications should be in the public domain) and some knowledge of the possible structure of the plaintext (such as the structure of a typical inter-bank financial transaction) for a partial or full reconstruction of the plaintext from ciphertext. Additionally, the goal is to also infer the key for decryption of future messages.

The precise methods used for cryptanalysis depend on whether the "attacker" has just a piece of ciphertext, or pairs of plaintext and ciphertext, how much structure is possessed by the plaintext, and how much of that structure is known to the attacker.

All forms of cryptanalysis for classical encryption exploit the fact that some aspect of the structure of plaintext may survive in the ciphertext.

key space: The total number of all possible keys that can be used in a cryptographic system. For example, **DES** uses a 56-bit key. So the key space is of size 2^{56}, which is approximately the same as 7.2×10^{16}.

brute-force attack: When encryption and decryption algorithms are publicly available, as they generally are, a brute-force attack means trying every possible key on a piece of ciphertext until an intelligible translation into plaintext is obtained.

codebook attack: The attacker tries to acquire as many as possible of the mappings between the plaintext words and the corresponding ciphertext words. In some cases, such a table, referred to as the codebook, can be used to speed up the brute-force search for the encryption key. As a trivial example, consider an 8-bit block cipher. If we can construct a codebook with 256 rows in it, that would break the cipher.

algebraic attack: You express the plaintext-to-ciphertext relationship as a system of equations. Given a set of (plaintext, ciphertext) pairs, you try to solve the equations for the encryption key. As you will see, encryption algorithms involve nonlinearities. In algebraic attacks, one attempts to introduce additional variables into the system of equations and make nonlinear equations look linear.

time-memory tradeoff in attacking ciphers: The brute-force and the codebook attacks represent two opposite cases in terms of time versus memory needs of the algorithms. Pure brute-force attacks have very little memory needs, but can require inordinately long times to scan through all possible keys. On the other hand, codebook attacks can in principle yield results instanta-

neously, but their memory needs can be humongously large. Just imagine a codebook for a 64-bit block cipher; it may need as many as 2^{64} rows in it. In some cases, by trading off memory for time, it is possible to devise more effective attacks that are sometimes referred to as *time-memory tradeoff attacks*. [As a specific example of time-memory tradeoff, we may be able to reduce the time taken by a brute-force attack if we use memory to store intermediate results obtained from the current computational steps (assuming they can help us avoid unnecessary search later during the computations). You will see examples of such tradeoffs in Lecture 24 when we talk about password cracking with rainbow tables.]

cryptology: Cryptography and cryptanalysis together constitute the area of cryptology

2.2: BUILDING BLOCKS OF CLASSICAL ENCRYPTION TECHNIQUES

- Two building blocks of all classical encryption techniques are **substitution** and **transposition**.

- Substitution means replacing an element of the plaintext with an element of ciphertext.

- Transposition means rearranging the order of appearance of the elements of the plaintext.

- Transposition is also referred to as permutation.

2.3: CAESAR CIPHER

- This is the earliest known example of a substitution cipher.

- Each character of a message is replaced by a character three position down in the alphabet.

```
plaintext:   are you ready

ciphertext:  DUH BRX UHDGB
```

- If we represent each letter of the alphabet by an integer that corresponds to its position in the alphabet, the formula for replacing each character p of the plaintext with a character c of the ciphertext can be expressed as

$$c \;=\; E(3,p) \;=\; (p+3)\; mod\; 26$$

where $E()$ stands for encryption. If you are not already familiar with modulo division, the *mod* operator returns the integer remainder of the division when $p+3$ is divided by 26, the number

of letters in the English alphabet. We are obviously assuming case-insensitive encoding with the Caesar cipher.

- A more general version of this cipher that allows for any degree of shift would be expressed by

$$c \;=\; E(k,p) \;=\; (p+k)\; mod\; 26$$

- The formula for decryption would be

$$p \;=\; D(k,c) \;=\; (c-k)\; mod\; 26$$

- In these formulas, k would be the secret key. As mentioned earlier, $E()$ stands for encryption. By the same token, $D()$ stands for decryption.

2.4: THE SWAHILI ANGLE ...

- A simple substitution cipher obviously looks much too simple to be able to provide any security, but that is the case only if you have some idea regarding the nature of the plaintext.

- What if the "plaintext" could be considered to be a binary stream of data and a substitution cipher replaced every consecutive 6 bits with one of 64 possible cipher characters? *In fact, this is referred to as Base64 encoding for sending email multimedia attachments.* [Did you know that all internet communications are character based? What does that mean and why do you think that is the case? What if you wanted to send a digital photo over the internet and one of the pixels in the photo had its graylevel value as 10 (hex: 0A)? If you put such a photo file on the wire without, say, Base64 encoding, why do you think that would cause problems? Imagine what would happen if you sent such a photo file to a printer without encoding. Visit `http://www.asciitable.com` to understand how the characters of the English alphabet are generally encoded. Visit the Base64 page at Wikipedia to understand why you need this type of encoding. A Base64 representation is created by carrying out a bit-level scan of the data and encoding it six bits at a time into a set of printable characters. For the most commonly used version of Base64, this 64-element set consists of the characters A-Z, a-z, 0-9, '+', and '/'.]

- If you did not know anything about the underlying plaintext and it was encrypted by a Base64 sort of an algorithm, it might not be as trivial a cryptographic system as it might seem. But, of course, if the word ever got out that your plaintext was in Swahili, you'd be hosed.

- Finally, here is more regarding the slogan "*All internet communications are character based*" in the red-and-blue note on the previous page: As you will see in Lecture 16, the internet communications are governed by the TCP/IP protocol. That protocol itself does not care whether you put on the wire a purely character based file, an audio file, a video file, etc. The protocol would work equally well with all sorts of files. So, strictly speaking, the slogan is technically wrong. Nonetheless, the slogan is of great practical importance because the software that is charged with the task of making your data file available to the TCP/IP engine in your computer could corrupt your data if it is not based on just printable characters.

2.5: A SEEMINGLY VERY STRONG MONOALPHABETIC CIPHER

- The Caesar cipher you just saw is an example of a **monoalphabetic cipher**. Basically, in a monoalphabetic cipher, you have a substitution rule that gives you a replacement ciphertext letter for each letter of the alphabet used in the plaintext message.

- Let's now consider what one would think would be a very strong monoalphabetic cipher. We will make our substitution letters a **random permutation** of the 26 letters of the alphabet:

```
plaintext letters:          a  b  c  d  e  f .....

substitution letters:       t  h  i  j  a  b .....
```

- The encryption key now is the sequence of substitution letters. In other words, the key in this case is the actual random permutation of the alphabet used.

- Since there are `26!` permutations of the alphabet, we end up with an extremely large key space. The number `26!` is much larger than 4×10^{26}. Since each permutation constitutes a key, that means that the monoalphabetic cipher has a key space of size larger than 4×10^{26}.

- Wouldn't such a large key space make this cipher extremely difficult to break? Not really, as we explain next!

2.5.1: A Very Large Key Space But ....

- The very large key space of a monoalphabetic cipher means that the total number of all possible keys that would need to be guessed in a pure brute-force attack would be much too large for such an attack to be feasible. (This key space is 10 orders of magnitude larger than the size of the key space for DES, the now somewhat outdated (but still widely used in the form of 3DES, as described in Lecture 9) NIST standard that is presented in Lecture 3.) [When you increase the size of a number by a factor of 10, you are increasing the size by *one order of magnitude*. So when we say that the keyspace is 10 orders of magnitude larger, that means that the keyspace is larger by a factor of 10^{10}. Recall, as mentioned in Section 2.1, the keyspace of DES is 2^{56} since the key size is 56 bits. And $2^{56} \approx 7.2 \times 10^{16}$.]

- Obviously, this would rule out a brute-force attack. Even if each key took only a nanosecond to try, it would still take zillions of years to try out even half the keys.

- So this would seem to be the answer to our prayers for an unbreakable code for symmetric encryption.

- But it is not! As to why? Read on.

2.6: THE ALL-FEARSOME STATISTICAL ATTACK

- If you know the nature of plaintext, any substitution cipher, regardless of the size of the key space, can be broken easily with a statistical attack.

- When the plaintext is plain English, a simple form of statistical attack consists measuring the frequency distribution for single characters, for pairs of characters, for triples of characters, and so on, and comparing those with similar statistics for English.

- Figure 1 shows the relative frequencies for the letters of the English alphabet in a sample of English text. Obviously, by comparing this distribution with a histogram for the letters occurring in a piece of ciphertext, you may be able to establish the true identities of the ciphertext letters.

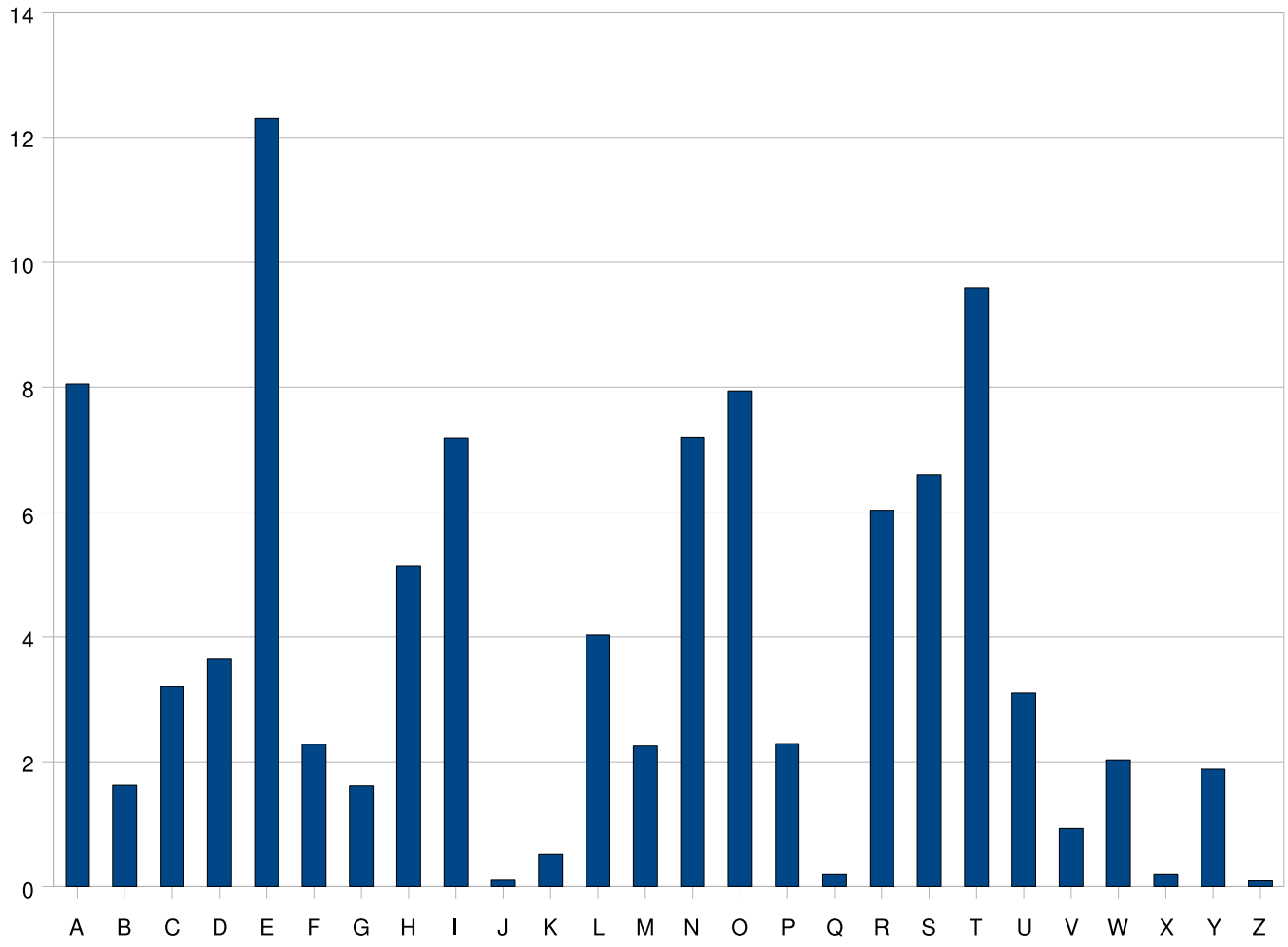

Figure 1: *Relative frequencies of occurrence for the letters of the alphabet in a sample of English text.* (*This figure is from Lecture 2 of "Computer and Network Security" by Avi Kak*)

2.6.1: Comparing the Statistics for Digrams and Trigrams

- Equally powerful statistical inferences can be made by comparing the relative frequencies for pairs and triples of characters in the ciphertext and the language believed to be used for the plaintext.

- Pairs of adjacent characters are referred to as **digrams**, and triples of characters as **trigrams**.

- Shown in Table 1 are the digram frequencies. The table does not include digrams whose relative frequencies are below 0.47. (A complete table of frequencies for all possible digrams would have 676 entries in it.)

- If we have available to us the relative frequencies for all possible digrams, we can represent this table by the joint probability $p(x, y)$ where x denotes the first letter of a digram and y the second letter. Such joint probabilities can be used to compare the digram-based statistics of ciphertext and plaintext.

- The most frequently occurring trigrams ordered by decreasing frequency are:

$$the \quad and \quad ent \quad ion \quad tio \quad for \quad nde \quad .....$$

digram	*frequency*	*digram*	*frequency*	*digram*	*frequency*	*digram*	*frequency*
th	3.15	to	1.11	sa	0.75	ma	0.56
he	2.51	nt	1.10	hi	0.72	ta	0.56
an	1.72	ed	1.07	le	0.72	ce	0.55
in	1.69	is	1.06	so	0.71	ic	0.55
er	1.54	ar	1.01	as	0.67	ll	0.55
re	1.48	ou	0.96	no	0.65	na	0.54
es	1.45	te	0.94	ne	0.64	ro	0.54
on	1.45	of	0.94	ec	0.64	ot	0.53
ea	1.31	it	0.88	io	0.63	tt	0.53
ti	1.28	ha	0.84	rt	0.63	ve	0.53
at	1.24	se	0.84	co	0.59	ns	0.51
st	1.21	et	0.80	be	0.58	ur	0.49
en	1.20	al	0.77	di	0.57	me	0.48
nd	1.18	ri	0.77	li	0.57	wh	0.48
or	1.13	ng	0.75	ra	0.57	ly	0.47

Table 1: *Digram frequencies in English text* (*This table is from Lecture 2 of "Computer and Network Security" by Avi Kak*)

2.7: MULTIPLE-CHARACTER ENCRYPTION TO MASK PLAINTEXT STRUCTURE: THE PLAYFAIR CIPHER

- One character at a time substitution obviously leaves too much of the plaintext structure in ciphertext.

- So how about destroying some of that structure by mapping multiple characters at a time to ciphertext characters?

- One of the best known approaches in classical encryption that carries out multiple-character substitution is known as the **Playfair cipher**, which is described in the next subsection.

2.7.1: Constructing the Matrix for Pairwise Substitutions in Playfair Cipher

In Playfair cipher, you first choose an encryption key. You then enter the letters of the key in the cells of a 5×5 matrix in a left to right fashion starting with the first cell at the top-left corner. You fill the rest of the cells of the matrix with the remaining letters in alphabetic order. The letters I and J are assigned the same cell. In the following example, the key is "**smythework**":

S	**M**	**Y**	**T**	**H**
E	**W**	**O**	**R**	**K**
A	*B*	*C*	*D*	*F*
G	*I/J*	*L*	*N*	*P*
Q	*U*	*V*	*X*	*Z*

2.7.2: Substitution Rules for Pairs of Characters in Playfair Cipher

1. Two plaintext letters that fall in the same row of the 5×5 matrix are replaced by letters to the right of each in the row. The "rightness" property is to be interpreted circularly in each row, meaning that the first entry in each row is to the right of the last entry. Therefore, the pair of letters "bf" in plaintext will get replaced by "CA" in ciphertext.

2. Two plaintext letters that fall in the same column are replaced by the letters just below them in the column. The "belowness" property is to be considered circular, in the sense that the topmost entry in a column is below the bottom-most entry. Therefore, the pair "ol" of plaintext will get replaced by "CV" in ciphertext.

3. Otherwise, for each plaintext letter in a pair, replace it with the letter that is in the same row but in the column of the other letter. Consider the pair "gf" of the plaintext. We have 'g' in the fourth row and the first column; and 'f' in the third row and the fifth column. So we replace 'g' by the letter in the same row as 'g' but in the column that contains 'f'. This given us 'P' as a replacement for 'g'. And we replace 'f' by the letter in the same row as 'f' but in the column that contains 'g'. That gives us 'A'

as replacement for ‘f’. Therefore, ‘gf’ gets replaced by ‘PA’.

2.7.3: Dealing with Duplicate Letters in a Key and Repeating Letters in Plaintext

- You must drop any duplicates in a key.

- Before the substitution rules are applied, you must insert a chosen "filler" letter (let's say it is 'x') between any repeating letters in the plaintext. So a plaintext word such as "hurray" becomes "hurxray"

2.7.4: How Secure is the Playfair Cipher?

- Playfair was thought to be unbreakable for many decades.

- It was used as the encryption system by the British Army in World War 1. It was also used by the U.S. Army and other Allied forces in World War 2.

- But, as it turned out, Playfair was extremely easy to break.

- As expected, the cipher does alter the relative frequencies associated with the individual letters and with digrams and with trigrams, but not sufficiently.

- Figure 2 shows the single-letter relative frequencies in descending order (and normalized to the relative frequency of the letter 'e') for some different ciphers. There is still considerable information left in the distribution for good guesses.

- The cryptanalysis of the Playfair cipher is also aided by the fact that a digram and its reverse will encrypt in a similar fashion.

That is, if AB encrypts to XY, then BA will encrypt to YX. So by looking for words that begin and end in reversed digrams, one can try to compare them with plaintext words that are similar. Example of words that begin and end in reversed digrams: receiver, departed, repairer, redder, denuded, etc.

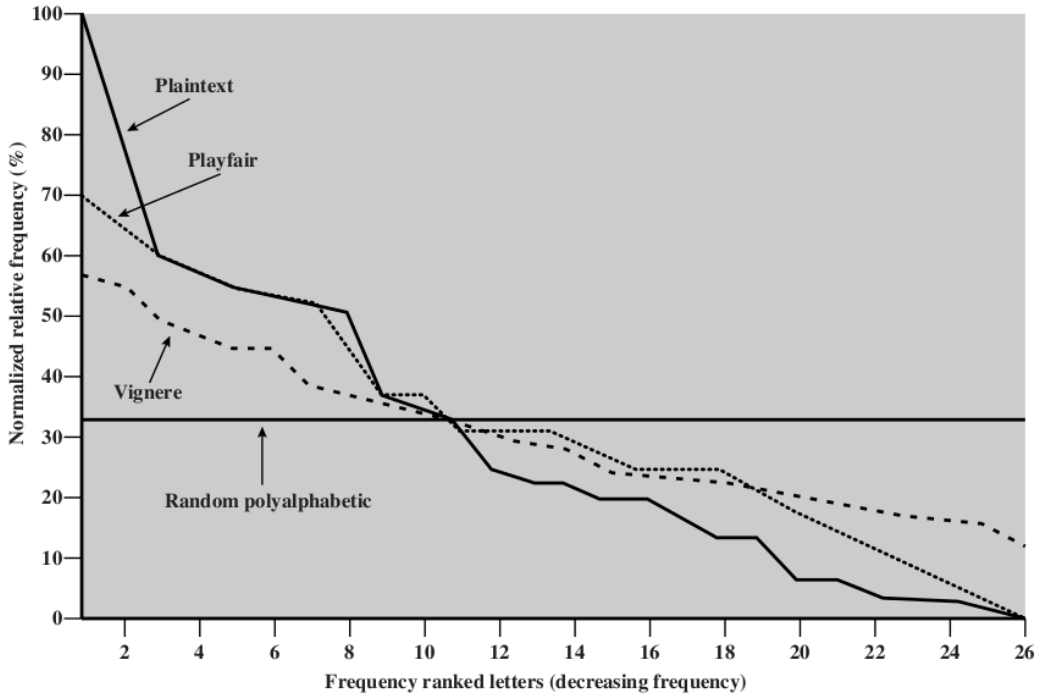

Figure 2: *Single-letter relative frequencies in descending order for a class of ciphers.* (*This figure is from Chapter 2 of William Stallings: "Cryptography and Network Security", Fourth Edition, Prentice-Hall.*)

2.8: ANOTHER MULTI-LETTER CIPHER: THE HILL CIPHER

- The Hill cipher takes a very different (more mathematical) approach to multi-letter substitution, as we describe in what follows.

- You assign an integer to each letter of the alphabet. For the sake of discussion, let's say that you have assigned the integers 0 through 25 to the letters 'a' through 'z' of the plaintext.

- The encryption key, call it $\mathbf{K}$, consists of a 3×3 matrix of integers:

$$\mathbf{K} = \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix}$$

- Now we can transform three letters at a time from the plaintext, the letters being represented by the numbers p_1, p_2, and p_3, into three ciphertext letters c_1, c_2, and c_3 in their numerical representations by

$$
\begin{aligned}
c_1 &= (k_{11}p_1 + k_{12}p_2 + k_{13}p_3) \; mod \; 26 \\
c_2 &= (k_{21}p_1 + k_{22}p_2 + k_{23}p_3) \; mod \; 26 \\
c_3 &= (k_{31}p_1 + k_{32}p_2 + k_{33}p_3) \; mod \; 26
\end{aligned}
$$

- The above set of linear equations can be written more compactly in the following vector-matrix form:

$$\vec{\mathbf{C}} = [\mathbf{K}]\,\vec{\mathbf{P}} \; mod \; 26$$

- Obviously, the decryption would require the inverse of $\mathbf{K}$ matrix.

$$\vec{\mathbf{P}} = \left[\mathbf{K}^{-1}\right]\vec{\mathbf{C}} \; mod \; 26$$

This works because

$$\vec{\mathbf{P}} = \left[\mathbf{K}^{-1}\right][\mathbf{K}]\,\vec{\mathbf{P}} \; mod \; 26 = \vec{\mathbf{P}}$$

2.8.1: How Secure is Hill Cipher?

- It is extremely secure against ciphertext only attacks. That is because the keyspace can be made extremely large by choosing the matrix elements from a large set of integers. (The key space can be made even larger by generalizing the technique to larger matrices.)

- But it has zero security when the plaintext–ciphertext pairs are known. The key matrix can be calculated easily from a set of known $\vec{\mathbf{P}}$, $\vec{\mathbf{C}}$ pairs.

2.9: POLYALPHABETIC CIPHERS: THE VIGENERE CIPHER

- In a monoalphabetic cipher, the same substitution rule is used at every character position in the plaintext message. In a polyalphabetic cipher, on the other hand, the substitution rule changes continuously from one character position to the next in the plaintext according to the elements of the encryption key.

- In the Vigenere cipher, you first "align" the encryption key with the plaintext message. [If the plaintext message is longer than the encryption key, you can repeat the encryption key, as we show below where the encryption key is "abracadabra".] Now consider each letter of the encryption key denoting a shifted Caesar cipher, the shift corresponding to the letter of the key. This is illustrated with the help of the table shown on the next page.

- Now a plaintext message may be encrypted as shown below:

```
key:          abracadabraabracadabraabracadabraab
plaintext:    canyoumeetmeatmidnightihavethegoods
ciphertext:   CBEYQUPEFKMEBK.....................
```

- The Vigenere cipher is an example of a polyalphabetic cipher.

- Since, in general, the encryption key will be shorter than the message to be encrypted, for the Vigenere cipher the key is repeated, as mentioned previously and as illustrated in the above example where the key is the string "abracadabra".

encryption key letter	*plain text letters* a	b	c	d	.....
	substitution letters				
a	A	B	C	D	.....
b	B	C	D	E	.....
c	C	D	E	F	.....
d	D	E	F	G	.....
e	E	F	G	H	.....
.	.	.	.	.	.
.	.	.	.	.	.
z	Z	A	B	C	.....

2.9.1: How Secure is the Vigenere Cipher?

- Since there exist in the output multiple ciphertext letters for each plaintext letter, you would expect that the relative frequency distribution would be effectively destroyed. But as can be seen in the plots in Figure 2, a great deal of the input statistical distribution still shows up in the output. [The plot shown for Vigenere cipher is for an encryption key that is just 9 letters long.]

- Obviously, the longer the encryption key, the greater the masking of the structure of the plaintext. The best possible key is as long as the plaintext message and consists of a purely random permutation of the 26 letters of the alphabet. This would yield the ideal plot shown in Figure 2. The ideal plot is labeled "Random polyalphabetic" in that figure.

- In general, to break the Vigenere cipher, you first try to estimate the length of the encryption key. This length can be estimated by using the logic that plaintext words separated by multiples of the length of the key will get encoded in the same way.

- If the estimated length of the key is N, then the cipher consists of

N monoalphabetic substitution ciphers and the plaintext letters at positions 1, N, 2N, 3N, etc., will be encoded by the same monoalphabetic cipher. This insight can be useful in the decoding of the monoalphabetic ciphers involved.

2.10: TRANSPOSITION TECHNIQUES

- All of our discussion so far has dealt with substitution ciphers. We have talked about monoalphabetic substitutions, polyalphabetic substitutions, etc.

- We will now talk about a different notion in classical cryptography: permuting the plaintext.

- This is how a pure permutation cipher could work: You write your plaintext message along the rows of a matrix of some size. You generate ciphertext by reading along the columns. The order in which you read the columns is determined by the encryption key:

```
key:                4 1 3 6 2 5

plaintext:          m e e t m e
                    a t m i d n
                    i g h t f o
                    r t h e g o
                    d i e s x y
```

```
ciphertext:          ETGTIMDFGXEMHHEMAIRDENOOYTITES
```

- The cipher can be made more secure by performing multiple rounds of such permutations.

2.11: Establishing Secure Communications for Fun (But Not for Profit)

- If your goal is to establish a medium-strength secure communication link, you may be able to get by without having to resort to the full-strength crypto systems that we will be studying in later lectures.

- This section presents two scripts, `EncryptForFun.py` and `DecryptForFun.py`, that you can use to create secure communication links with your friends and relatives. Fundamentally, the encryption/decryption logic in these scripts is based on the following properties of XOR operations on bit blocks. Assuming that A, B, and C are bit arrays, we can write

$$\begin{aligned} [A \oplus B] \oplus C &= A \oplus [B \oplus C\ ] \\ A \oplus A &= 0 \\ A \oplus 0 &= A \end{aligned}$$

- More precisely, the encryption script shown below is based on differential XORing of bit blocks. The document to be encrypted

is scanned in bit blocks of size `BLOCKSIZE`. Let the bit blocks be denoted $B0, B1, B2, \ldots$. After it is XORed with the key, the very first bit block, $B0$, is XORed with an initialization vector (IV) that is derived from a pass-phrase. The output of this operation is XORed with the key-XORed $B1$, and so on.

- Differential XORing destroys any repetitive patterns in the messages to be encrypted and makes it more difficult to break encryption by statistical analysis. Differential XORing needs an Initialization Vector that, as already mentioned, is derived from a pass phrase in the script shown below.

- The implementation shown below is made fairly compact by the use of the `BitVector` module. [**This would be a good time to become familiar with the `BitVector` module by going through its API. You'll be using this module in several homework assignments dealing with cryptography and hashing.**]

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

###  EncryptForFun.py
###  Avi Kak  (kak@purdue.edu)
###  January 21, 2014

###  Medium strength encryption/decryption for secure
###  message exchange for fun.

###  Call syntax:
###
```

```
###       EncryptForFun.py  message_file.txt  output.txt
###
###  The encrypted output is deposited in the file 'output.txt'


PassPhrase = "Hopes and dreams of a million years"

import sys
from BitVector import *                                          #(A)

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

BLOCKSIZE = 64                                                   #(C)
numbytes = BLOCKSIZE / 8                                         #(D)

# Reduce the passphrase to a bit array of size BLOCKSIZE:
bv_iv = BitVector(bitlist = [0]*BLOCKSIZE)                       #(E)
for i in range(0,len(PassPhrase) / numbytes):                    #(F)
    textstr = PassPhrase[i*numbytes:(i+1)*numbytes]              #(G)
    bv_iv ^= BitVector( textstring = textstr )                   #(H)

# Get key from user:
try:                                                             #(I)
    key = raw_input("Enter key: ")                               #(J)
except EOFError: sys.exit()                                      #(K)
if len(key) < numbytes:                                          #(L)
    key = key + '0' * (numbytes-len(key))                        #(M)

# Reduce the key to a bit array of size BLOCKSIZE:
key_bv = BitVector(bitlist = [0]*BLOCKSIZE)                      #(N)
for i in range(0,len(key) / numbytes):                           #(O)
    keyblock = key[i*numbytes:(i+1)*numbytes]                    #(P)
    key_bv ^= BitVector( textstring = keyblock )                 #(Q)

# Create a bitvector for storing the ciphertext bit array:
msg_encrypted_bv = BitVector( size = 0 )                         #(R)

# Carry out differential XORing of bit blocks and encryption:
previous_block = bv_iv                                           #(S)
bv = BitVector( filename = sys.argv[1] )                         #(T)
while (bv.more_to_read):                                         #(U)
```

```
        bv_read = bv.read_bits_from_file(BLOCKSIZE)                    #(V)
        if len(bv_read) < BLOCKSIZE:                                   #(W)
            bv_read += BitVector(size = (BLOCKSIZE - len(bv_read)))    #(X)
        bv_read ^= key_bv                                              #(Y)
        bv_read ^= previous_block                                      #(Z)
        previous_block = bv_read.deep_copy()                           #(a)
        msg_encrypted_bv += bv_read                                    #(b)
    outputhex = msg_encrypted_bv.getHexStringFromBitVector()           #(c)

    # Write ciphertext bitvector to the ouput file:
    FILEOUT = open(sys.argv[2], 'w')                                   #(d)
    FILEOUT.write(outputhex)                                           #(e)
    FILEOUT.close()                                                    #(f)
```

- In the script shown above, if the size (in terms of the number of bits) of the message file is not an integral multiple of `BLOCKSIZE`, the script appends a sequence of null bytes (that is, bytes made up of all zeros) at the end so that this condition is satisfied. This is done in line (W) and (X) of the script.

- The decryption script, shown below, uses the same properties of the XOR operator as stated at the beginning of this section to recover the original message from the encrypted output.

- The reader may wish to compare the decryption logic in the loop in lines (U) through (b) of the script shown below with the encryption logic shown in lines (S) through (b) of the script above.

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

###  DecryptForFun.py
###  Avi Kak  (kak@purdue.edu)
###  January 21, 2014

###  Medium strength encryption/decryption for secure
###  message exchange for fun.

###  Call syntax:
###
###        DecryptForFun.py  encrypted_file.txt  recover.txt
###
###  The decrypted output is deposited in the file 'recover.txt'

PassPhrase = "Hopes and dreams of a million years"

import sys
from BitVector import *                                          #(A)

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

BLOCKSIZE = 64                                                   #(C)
numbytes = BLOCKSIZE / 8                                         #(D)

# Reduce the passphrase to a bit array of size BLOCKSIZE:
bv_iv = BitVector(bitlist = [0]*BLOCKSIZE)                       #(E)
for i in range(0,len(PassPhrase) / numbytes):                    #(F)
    textstr = PassPhrase[i*numbytes:(i+1)*numbytes]              #(G)
    bv_iv ^= BitVector( textstring = textstr )                   #(H)

# Create a bitvector from the ciphertext hex string:
FILEIN = open(sys.argv[1])                                       #(I)
encrypted_bv = BitVector( hexstring = FILEIN.read() )            #(J)

# Get key from user:
try:                                                             #(K)
    key = raw_input("Enter key: ")                               #(L)
except EOFError: sys.exit()                                      #(M)
if len(key) < numbytes:                                          #(N)
    key = key + '0' * (numbytes-len(key))                        #(O)
```

```
# Reduce the key to a bit array of size BLOCKSIZE:
key_bv = BitVector(bitlist = [0]*BLOCKSIZE)                          #(P)
for i in range(0,len(key) / numbytes):                               #(Q)
    keyblock = key[i*numbytes:(i+1)*numbytes]                        #(R)
    key_bv ^= BitVector( textstring = keyblock )                     #(S)

# Create a bitvector for storing the output plaintext bit array:
msg_decrypted_bv = BitVector( size = 0 )                             #(T)

# Carry out differential XORing of bit blocks and decryption:
previous_decrypted_block = bv_iv                                     #(U)
for i in range(0, len(encrypted_bv) / BLOCKSIZE):                    #(V)
    bv = encrypted_bv[i*BLOCKSIZE:(i+1)*BLOCKSIZE]                   #(W)
    temp = bv.deep_copy()                                            #(X)
    bv ^=  previous_decrypted_block                                  #(Y)
    previous_decrypted_block = temp                                  #(Z)
    bv ^=  key_bv                                                    #(a)
    msg_decrypted_bv += bv                                           #(b)

outputtext = msg_decrypted_bv.getTextFromBitVector()                 #(c)

# Write the plaintext to the output file:
FILEOUT = open(sys.argv[2], 'w')                                     #(d)
FILEOUT.write(outputtext)                                            #(e)
FILEOUT.close()                                                      #(f)
```

- To exercise these scripts, enter some text in a file and let's call this file `message.txt`. Now you can call the encrypt script by

  ```
  EncryptForFun.py  message.txt  output.txt
  ```

  The script will place the encrypted output, in the form of a hex string, in the file `output.txt`. Subsequently, you can call

  ```
  DecryptForFun.py  output.txt   recover.txt
  ```

  to recover the original message from the encrypted output produced by the first script.

- The security level of this script can be taken to full strength by using 3DES or AES for encrypting the bit blocks produced by differential XORing.

2.12: HOMEWORK PROBLEMS

1. Use the ASCII codes available at `http://www.asciitable.com` to manually construct a Base64 encoded version of the string "`hello\njello`". Your answer should be "`aGVsbG8KamVsbG8=`". What do you think the character '=' at the end of the Base64 representation is for? [If you wish you can also use interactive Python for this. Enter the following sequence of commands "`import base64`" followed by "`base64.b64encode('hello\njello')`". If you are using Python 3, make sure you prefix the argument to the `b64encode()` function by the character 'b' to indicate that it is of type `bytes` as opposed to of type `str`. Several string processing functions in Python 3 require `bytes` type arguments and often return results of the same type. Educate yourself on the difference between the string `str` type and `bytes` type in Python 3.]

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

   ```
   hello
   ```

   If you examine this file with a command like

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

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

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

which translate into the following bit content:

```
01101000 01100101 01101100 01101100 01101111 00001010
```

Looks like there are six bytes in the file whereas the word "hello" has only five characters. What do you think is going on? Do you know why your editor might want to place that extra byte in the file and how to prevent that from happening?

3. All classical ciphers are based on symmetric key encryption. What does that mean?

4. What are the two building blocks of all classical ciphers?

5. True or false: The larger the size of the key space, the more secure a cipher? Justify your answer.

6. Give an example of a cipher that has an extremely large key space size, an extremely simple encryption algorithm, and extremely poor security.

7. What is the difference between monoalphabetic substitution ciphers and polyalphabetic substitution ciphers?

8. What is the main security flaw in the Hill cipher?

9. What makes Vigenere cipher more secure than, say, the Playfair cipher?

10. **Programming Assignment:**

    Write a script called `hist.pl` in Perl (or `hist.py` in Python) that makes a histogram of the letter frequencies in a text file. The output should look like

```
A: xx
B: xx
C: xx
...
...
```

    where xx stands for the count for that letter.

11. **Programming Assignment:**

    Write a script called `poly_cipher.pl` in Perl (or `poly_cipher.py` in Python) that is an implementation of the Vigenere polyalphabetic cipher for messages composed from the letters of the English alphabet, the numerals 0 through 9, and the punctuation marks '.', ',', and '?'.

    Your script should read from standard input and write to standard output. It should prompt the user for the encryption key.

    Your hardcopy submission for this homework should include some sample plaintext, the ciphertext, and the encryption key used.

Make your scripts as compact and as efficient as possible. Make liberal use of builtin functions for what needs to be done. For example, you could make a circular list with either of the following two constructs in Perl:

```
unshift( @array, pop(@array) )
push( @array, shift(@array) )
```

See perlfaq4 for some tips on array processing in Perl.

12. **Programming Assignment:**

This is an exercise in you assuming the role of a cryptanalyst and trying to break a cryptographic system that consists of the two Python scripts you saw in Section 2.11. As you'll recall, the script `EncryptForFun.py` can be used for encrypting a message file and the script `DecryptForFun.py` for recovering the plaintext message from the ciphertext created by the first script. You can download both these scripts in the code archive for Lecture 2.

With `BLOCKSIZE` set to 16, the script `EncryptForFun.py` produces the following ciphertext output for a plaintext message that is a quote by Mark Twain:

```
20352a7e36703a6930767f7276397e376528632d6b6665656f6f6424623c2d\
30272f3c2d3d2172396933742c7e233f687d2e32083c11385a03460d440c25
```

all in one line. (You can copy-and-paste this hex ciphertext into your own script. However, make sure that you delete the back-slash at the end of the first line. You can also see the same

output in the file named `output5.txt` in the code archive for Lecture 2.) Your job is to both recover the original quote and the encryption key used by mounting a brute-force attack on the encryption/decryption algorithms. (**HINT:** The logic used in the scripts implies that the effective key size is only 16 bits when the `BLOCKSIZE` variable is set to 16. So your brute-force attack need search through a keyspace of size only 2^{16}.)

CREDITS

The data presented in Figure 1 and Table 1 are from `http://jnicholl.org/Cryptanalysis/Data/EnglishData.php`. That site also shows a complete digram table for all 676 pairings of the letters of the English alphabet.