CS342 Computer Security | **Handout # 3**
Profs. Daniel Bilar and Lyn Turbak | **September 12, 2006**
Wellesley College

Problem Set 1

Due: Midnight Friday, September 22

Overview:

The purpose of this assignment is to give you experience with basic cryptography and Linux system administration.

Working Together:

There are three problems on this assignment:

1. Problem 1 is an individual problem that you must solve on your own.
2. Problem 2 is a group problem that you *may* work on in groups of up to three people. It is *strongly* recommended that you work in a large group. Your partners for this problem need not be the same as for Problem 3 – indeed, it would be good to choose different partners for Problems 2 and 3.
3. Problem 3 is a lab problem in which you will become the system administrators for a Linux machine in 121B. You *must* work in pairs on this problem; because there are an odd number of students in the class, we will also allow one three-person team. Students with a strong systems background should *not* work with each other, but should choose partners with less systems background.

Submission:

1. For Problem 1, each student should submit a hardcopy of a solution by sliding it under Lyn's door.
2. For Problem 2, each group should submit a softcopy to the drop directory

   `~cs342/drop/ps1/`*username*

   where *username* is the username of one of the group members. The softcopy should include: (1) six files `m1.txt`–`m6.txt` that are the decoded versions of the binary files `m1.bin`–`m6.bin` from the problem; and (2) a description of how you paired the files and decoded them. If you are not able to decode all the files, turn in whatever work you have done, including partially decoded files and a description of your experiments.
3. For Problem 3, there is nothing to turn in, but Daniel and Lyn will check that the `guest` account is configured appropriately on your machine.

Individual Problem [15]: Cryptography with Unshared Keys

This is an individual problem. Each student must solve this problem on her own without consulting any other person (except Daniel and Lyn).

Alice and Bob have been using symmetric-key cryptography to protect the confidentiality of their communications, but they think that exchanging shared keys is a hassle.

Inspired by the bicycle transfer protocol that will be discussed in Sep. 12's class[1], they have developed their own **unshared-key** protocol that allows them to communicate without exchanging a shared key. In their new protocol, here's how Alice sends a message M to Bob:

1. Alice chooses a new key K_A that is the same length as M, and sends $M_2 = M \oplus K_A$ to Bob.
2. Bob chooses a new key K_B that is the same length as M_2, and sends $M_3 = M_2 \oplus K_B$ to Alice.
3. Alice sends $M_4 = M_3 \oplus K_A$ to Bob.
4. Bob calculates $M_4 \oplus K_B$ to extract M.

Alice and Bob reason that K_A and K_B are like their own personal locks in the bicycle transfer protocol, so they never have to meet to exchange keys.

a. Using algebra involving $\oplus$, show that the protocol does indeed send M from Alice to Bob. Recall that $\oplus$ has the following algebraic properties:

associativity: $X \oplus (Y \oplus Z) = (X \oplus Y) \oplus Z$
commutativity: $X \oplus Y = Y \oplus X$
identity: $X \oplus 0 = X$
invertibility: $X \oplus X = 0$

You should show that $M_4 \oplus K_B$ simplifies to M, explicitly using the above properties to justify each step of your simplification.

b. Show that Alice can determine Bob's key and Bob can determine Alice's key. Does this make the protocol insecure?

c. Suppose Eve is monitoring all the communications between Alice and Bob. Can she determine the message M? Explain.

[1]You can solve this problem before you see the bicycle transfer protocol.

Group Problem 1 [75]: Two-Timing Pads

This is a group problem. Each student may solve this problem in a group of up to three people.

Until she resigned recently, Malificent (Mali) Cracker was an employee at Mad Hatters, Inc., a computer security startup in the Boston area founded by former CS342 students. Mad Hatters' Chief Security Officer, Benny Hacker, suspects that Mali was a spy from JetCap, a rival company, and that she used the company's email system to send encrypted messages to JetCap.

Benny suspects that Mali used one-time pads to encrypt her messages. When used correctly, a one-time pad provides perfect confidentiality. However, Benny also knows that Mali tended cut corners and suspects that she used her pads more than once — a practice that could be her downfall.

Because he trusts CS342 students, Benny has hired you as interns to help him uncover Mali's perfidy. Benny has collected six of the encrypted messages he intercepted from Mali's emails and put the first 512 bytes of each message into the files `m1.bin` through `m6.bin` in the CS342 download directory: `cs342@cs.wellesley.edu:/home/cs342/download/ps1`. He believes that these six files were encrypted using three different pads, where each pad was used to encrypt two files. Your task is to figure out which files are paired and to decrypt all of the files.

Benny has developed some programs that will be useful for your project; these can also be found in the above directory. He has described these in a memo, which is included as Appendix B at the end of this problem set. You should read his memo carefully before proceeding.

Benny has also collected the following information from reliable sources:

- One of the messages is a Java program.
- One of the messages is a web page expressed in HTML.
- Each pair of messages includes one message that is either a quotation from literature or lyrics from a song. So if you figure out a few key words from the message, you can use a Google search to flesh out the rest of the message.[2]
- Mali had an unusual level of interest in the Wellesley Computer Science department. She liked music and TV shows from the 1960's. She also liked to hike and swim at Walden pond.

Hints

- Assume that all six plaintext files contain only "regular" characters whose ASCII values only include tab (9) newline (10) and characters in the range 32 − 126. In particular, the highest-order bit of each such character is 0. (See `http://www.asciitable.com`.)

- You can determine which pairs of files were encrypted with the same pad just using `cat` in conjunction with Benny's `xor` program. How?

- Benny's `check` program should be your main tool for decoding the messages. Decoding with `check` will require lots of trial and error. You can reduce your time by thinking carefully about what you're looking for.

- Java and HTML files have a lot of structure. Using Benny's `check` program, how can you determine which pair contains the Java program? the HTML page?

- It's helpful to look for common English words, like and, `have`, `that`, `the`, `which`, `will`, `with`. It often helps to put a space before and after such words — e.g., search for `" and "` rather than `"and"`.

- You shouldn't need to write any additional programs, but you can if you want to.

[2]In practice, it would be unusual for a spy to send a message whose contents could be easily found on the Web. Why not just send the URL instead? But here, this fact simplifies your problem tremendously.

Group Problem 2 [10]: Linux System Administration

Please do not start this problem until Daniel and Lyn say it's OK to do so.

*You *must* work in a group on this problem with your Linux buddy. You will work with the same Linux buddy all semester. You need not work with the same person as on Problem 2.*

In this problem, you will choose a buddy and a machine in the Security Lab (SCI 121B) and become system adminstrators (sysadmins) of that machine. You and your buddy will "own" that machine for the rest of the semester.

There are seven machines in SCI 121B that are "named" by their static IP addresses, which are in the range `192.168.0.1` – `192.168.0.7`. We will refer to this network of machines as the *security lab network (SLN)*. Since the SLN has no nameserver, the machines will not recognize the English names `carp`, `sole`, etc., that appear on the processor boxes; you will have to use the IP numbers to refer to other machines.

Below, we will use the last number in each address (in the range 1–7) to uniquely identify a machine.

There are three kinds of machines:

- #1 is a firewall that interfaces between the other six machines and the main network.
- #2 is the instructor machine, reserved for Daniel and Lyn.
- #3–#7 are reserved for students.

a. : Choose a Machine

Choose one of machines #3–#7 in 121B, putting a note on it to indicate it is "yours".

b. : Log into Your Machine

You can log into your machine with superuser privileges via the username `root` and password `toober`.

c. : Change the Root Password

You wouldn't want anyone else to log into *your* machine with superuser privileges, would you? One of the first things you should do is change the root password using the `passwd` program.

d. : Create A Guest Account

Next, you should use the `useradd` program to create a `guest` account with password `"Ewe guessed it!"` (without the quote marks). You can set the password for this account using the `passwd` program and/or the `-p` option to the `useradd` or `usermod` programs. If you wish, you can create accounts other than the guest account.

e. : Play with Linux

As system administrators, you need to gain familiarity with lots of Linux commands. Appendix A lists some of the commands you should be familiar with. The list is by no means exhaustive! Many of the commands have lots of options; some of the common options ones are listed. To get documentation on this commands, use `man` and `info`, browse on-line resources, and refer to the many Linux books in the Security Lab. You should also play with pipes (`|`) and input/output redirection (`<`, `>`) and learn a little bit about shell scripts.

Appendix A: Linux Commands you Should Know

```
cat
cd
chmod (-R)
chown (-R)
chgrp (-R)
cp (-R)
du
df
echo
find
grep
gunzip
gzip
info
less
ln (-s)
ls
man
mkdir
more
mount
nice
passwd
popd
ps (-ef)
pushd
pwd
rm (-rf)
scp (-r)
ssh
source
su (-)
tar (-cvf, -xvf)
telnet
top
touch
umount
useradd
usermod
wc
which
whoami
```

Appendix B: Benny's Memo

Character Representations

To understand encryption/decryption with one-time pads, you first need to understand character representations. Recall that characters are often expressed as 8-bit bytes with ASCII values ranging between 0 and 255 (see `http://www.asciitable.com`). Fig. 1 shows the printed representations of these characters as they appear in an Emacs file buffer. Because character representations are not consistent across different environments, I recommend that you always view arbitrary binary files within Emacs to get results that resemble the ones you see in this memo.

The figure was obtained[3] by executing the following C[4] program and storing the results in a file:

```
int main () {
  int i;
  for (i = 0; i<=255; i++) {
    printf("%d:%c\t", i, (char) i);
    if (i%4 == 3) printf("\n");
  }
}
```

Some features of Fig. 1 deserve explanation:

- Characters in ASCII range 0 – 31 and ASCII 127 are *control characters.* Most are represented as a caret symbol (^) (pronounced "control") followed by another character. But a few actually cause interesting effects in the text:
  - ASCII 9 (control-I) is the tab character;
  - ASCII 10 (control-J) is the newline character, which moves to the next line;
- ASCII characters c in the range 128–255 are represented as `\ddd`, where `ddd` is the octal representation of c. For example, ASCII value $205 = 3 \cdot 8^3 + 1 \cdot 8^1 + 5 \cdot 8^0$, so it is represented ad `\315`.
- We will use the term *regular text characters* for characters that are the tab character (ASCII value 9), the newline character (ASCII value 10), and characters in the ASCII range 32–126. All regular text characters have 0 as their high bit, so they are completely determined by their lower 7 bits.

[3]But some mopping up was done to make the columns better aligned.

[4]In CS342, you'll need to have a working understanding of C programming, which you're expected to "pick up" largely on your own (but be sure to ask your instructors any questions you have along the way). A good place to start is Scott Anderson's *C/C++ for Java Programmers*, linked from the CS342 home page.

0:^@	1:^A	2^B	3:^C	4:^D	5:^E	6:^F	7:^G
8:^H	9:	10:					
	11:^K	12:^L	13:^M	14:^N	14:^O		
16:^P	17:^Q	18:^R	19:^S	20:^T	21:^U	22:^V	23:^W
24:^X	25:^Y	26:^Z	27:^[	28:^\	29:^]	30:^^	31:^_
32:	33:!	34:"	35:#	36:$	37:%	38:&	39:’
40:(	41:)	42:*	43:+	44:,	45:-	46:.	47:/
48:0	49:1	50:2	51:3	52:4	53:5	54:6	55:7
56:8	57:9	58::	59:;	60:<	61:=	62:>	63:?
64:@	65:A	66:B	67:C	68:D	69:E	70:F	71:G
72:H	73:I	74:J	75:K	76:L	77:M	78:N	79:O
80:P	81:Q	82:R	83:S	84:T	85:U	86:V	87:W
88:X	89:Y	90:Z	91:[	92:\	93:]	94:^	95:_
96:‘	97:a	98:b	99:c	100:d	101:e	102:f	103:g
104:h	105:i	106:j	107:k	108:l	109:m	110:n	111:o
112:p	113:q	114:r	115:s	116:t	117:u	118:v	119:w
120:x	121:y	122:z	123:{	124:|	125:}	126:~	127:^?
128:\200	129:\201	130:\202	131:\203	132:\204	133:\205	134:\206	135:\207
136:\210	137:\211	138:\212	139:\213	140:\214	141:\215	142:\216	143:\217
144:\220	145:\221	146:\222	147:\223	148:\224	149:\225	150:\226	151:\227
152:\230	153:\231	154:\232	155:\233	156:\234	157:\235	158:\236	159:\237
160:\240	161:\241	162:\242	163:\243	164:\244	165:\245	166:\246	167:\247
168:\250	169:\251	170:\252	171:\253	172:\254	173:\255	174:\256	175:\257
176:\260	177:\261	178:\262	179:\263	180:\264	181:\265	182:\266	183:\267
184:\270	185:\271	186:\272	187:\273	188:\274	189:\275	190:\276	191:\277
192:\300	193:\301	194:\302	195:\303	196:\304	197:\305	198:\306	199:\307
200:\310	201:\311	202:\312	203:\313	204:\314	205:\315	206:\316	207:\317
208:\320	209:\321	210:\322	211:\323	212:\324	213:\325	214:\326	215:\327
216:\330	217:\331	218:\332	219:\333	220:\334	221:\335	222:\336	223:\337
224:\340	225:\341	226:\342	227:\343	228:\344	229:\345	230:\346	231:\347
232:\350	233:\351	234:\352	235:\353	236:\354	237:\355	238:\356	239:\357
240:\360	241:\361	242:\362	243:\363	244:\364	245:\365	246:\366	247:\367
248:\370	249:\371	250:\372	251:\373	252:\374	253:\375	254:\376	255:\377

Figure 1: A table showing the representation of characters in an Emacs file buffer.

Suppose that the file containing the characters in Fig. 1 is named `chartable.bin`.[5] Then using the Linux `cat` command to display the contents of this file in an Emacs shell[6] yields a result that is *almost* like Fig. 1 except for the first few lines:

```
[cs342@puma] cat chartable.bin
0:^@        1:^A        2^B         3:^C        4:^D        5:^E        6:^F        7:^G
8       9:          10:
        14:^N       15:^O
16:^P       17:^Q       18:^R       19:^S       20:^T       21:^U       22:^V       23:^W
```

The differences are due to the fact that the Emacs shell performs actions for some of the control characters:

- ASCII 8 (control-H) is treated as backspace, so it erases the colon after the 8;
- ASCII 13 (control-M) is treated as a carriage return, which erases all that precedes it on the current line (in this case, `11:^K        12:^L        13:`).

I have written a utility program name `showall` that shows printed representations for all of the characters, including `^H`, `^I`, `^J`, and `^M`. For example:

```
[cs342@puma] cat chartable.bin | head | showall
0:^@^I1:^A^I2:^B^I3:^C^I4:^D^I5:^E^I6:^F^I7:^G^I^J8:^H^I9:^I^I10:^J^I11:^K^I12:^L
^I13:^M^I14:^N^I15:^O^I^J16:^P^I17:^Q^I18:^R^I19:^S^I20:^T^I21:^U^I22:^V^I23:^W^I
^J24:^X^I25:^Y^I26:^Z^I27:^[^I28:^\^{}I29:^]^I30:^^^I31:^_^I^J32: ^I33:!^I34:"^I35:
#^I36:$^I37:%^I38:&^I39:'^I^J40:(^I41:)^I42:*^I43:+^I44:,^I45:-^I46:.^I47:/^I^J48
:0^I49:1^I50:2^I51:3^I52:4^I53:5^I54:6^I55:7^I^J56:8^I57:9^I58::^I59:;^I60:<^I61:
=^I62:>^I63:?^I^J64:@^I65:A^I66:B^I67:C^I68:D^I69:E^I70:F^I71:G^I^J
```

(The Linux `head` command returns the first ten lines of a file.)

Encrypting/Decrypting with Pads

A one-time pad of length n is a sequence of n random characters. Fig. 2 presents a C program that approximates a one-time pad using a pseudorandom number generator with a seed determined by the system clock.

Suppose the program is named `pad`. Here is an example where `>` redirects the output of `pad` into a file named pad200.bin for later use and `cat` is used to display the contents of this file:

```
[cs342@puma] pad 200 > pad200.bin
[cs342@puma] cat pad200.bin | showall
^E\374\253c\275hw^K \212\221\357\337C\251\261\354[^DC]9\203S*\362,\367\274g^O\301
c\273$ #\233+C%\2743^D\377\334\265\3538\272/\225\363\262\350^]\244^T^U`{$"\337\33
7F\377^B\342+F^G\347y^L\347U\301\322\215{^A"n\263^K\214X^_\241\270\233\305\332z\2
45!y\247^C\244\355^J\214f^Vs\274\330EISGl\302\372wNR\226\357^K1\264\345\253Y^F%^A
^I\311\356^TUU*\310^Q^B^NZVU\306^XO=f\242\324U\255^E^I\222\261c\231\326d\242\237R
\266\365\247\341\275\270\343\313^S9 \331Qp^W\267^R\353^L\277\360^VQ\241y\352w\335
\215^W/C^L\327$\311
```

A file can be encrypted/decrypted with a pad by XOR-ing the characters in the file with those in the pad. I have written an `xor` program (Fig. 3) for XORing two files that can be used for this purpose.

[5] We will use the convention that files with the `.bin` extension contain arbitrary characters while those with the `.txt` extension contain only regular text characters.

[6] To start a shell within Emacs, use `M-x shell`. All shell examples in this memo are from an Emacs shell. You will see different behavior if you use a shell outside of Emacs, so executing all shell commands within an Emacs shell is recommended.

```
/* Benny's program to illustrate pad generation.
   It writes to standard output a psuedorandom sequence of N
   characters, where N is specified as the program argument.
   The seed for the psuedorandom generator is the current time.

   Since all the generator's information is in a small, and possibly
   guessable, seed, this is *not* an effective way to generate
   a one-time pad. But it is helpful for illustrating how
   such pads are used. */

#include <stdio.h>  /* Include standard I/O constants and operations */
#include <stdlib.h> /* Include standard library, which includes pseudorandom
                       number generator */
#include <time.h>   /* Include time functions for initializing seed
                       of pseudorandom number generator */

/* Generate a random character */
char randomChar() {
  /* Take remainder of random number relative to 256
     (A character in C is an unsigned int in range 0--255) */
  return (rand() % 256);
}

int main (int argc, char* argv[]) {/* argc = number of args;
                                      argv[0] = program name;
                                      argv[1] = int specifying number of chars */
  if (argc != 2) { /* complain if unexpected number of arguments */
    printf("%s expects one argument", argv[0]);
    exit(1); /* terminate program with error code */
  }
  int n = atoi(argv[1]); /* store integer representation of arg in n;
                            stores 0 for string in which no prefix is an int. */
  if (n < 0) {
    printf("%s expects a non-negative integer but %d is not", argv[0], n);
    exit(1); /* terminate program with error code */
  }
  int i;
  srand((unsigned) time (NULL)); /* initialize seed of pseudorandom number generator
                                    using the current time */
  for (i = 0; i < n; i++) {
    printf("%c", randomChar()); /* write random character to stdout */
  }
  exit(0); /* terminate program with normal code */
}
```

Figure 2: A C program generating an approxiation to a one-time pad.

```
/* Benny's program that XORs the contents of two files.
   It writes to stdout the XOR on the character contents of two files:
      (1) stdin
      (2) the file named by the single program argument.
   If one file is shorter than the other, ignores the characters
   of the longer file. */

#include <stdio.h>  /* Include standard I/O constants and operations */

int main (int argc, char* argv[]) {/* argc = number of args;
                                      argv[0] = program name;
                                      argv[1] = filename */
  FILE* f;     /* holds pointer to file */
  int i1, i2; /* character variables */
  if (argc != 2) { /* complain if unexpected number of arguments */
    printf("%s expects one argument", argv[0]);
    exit(1); /* terminate program with error code */
  }
  f = fopen(argv[1], "r"); /* open named file for reading */
  if (f == NULL) { /* complain if file not found */
    printf("File %s not found!\n", argv[1]);
    exit(1); /* terminate program with error code */
  }
  i1 = fgetc(stdin); /* read first character from stdin */
  i2 = fgetc(f);     /* read first character from file */
  /* printf("| c1 = %c (%d); c2 = %c (%d);\n", i1, i1, i2, i2); */
  while ((i1 != EOF) && (i2 != EOF)) { /* while both files nonempty, */
    /* printf("c1^c2 = %c (%d)\n", (char) (i1^i2), (char) (i1^i2));  */
    printf("%c", (char) (i1^i2)); /* write xor of corresponding characters to stdout */
    i1 = fgetc(stdin);   /* read next character from stdin */
    i2 = fgetc(f);       /* read next character from file */
    /*printf("| c1 = %c (%d); c2 = %c (%d);\n", i1, i1, i2, i2);*/
  }
  /* printf("\n"); */ /* Print a newline to flush print buffer */
  exit(0); /* terminate program with normal code */
}
```

Figure 3: A C program for XORing two files.

For example, suppose that `tiger.txt` contains the following poem from Kurt Vonnegut's book *Cat's Cradle*:

```
[cs342@puma ps1] cat tiger.txt
Tiger got to hunt,
Bird got to fly;
Man got to sit and wonder, "Why, why, why?"

Tiger got to sleep,
Bird got to land;
Man got to tell himself he understand.
```

The Linux word count program `wc` tells us that `tiger.txt` has 7 lines, 32 words and 158 characters:

```
[cs342@puma] wc tiger.txt
      7      32     158 tiger.txt
```

We can use the `xor` program to combine `tiger.txt` with the pad in `pad200.bin` to create the file `tiger.bin`:

```
[cs342@puma] cat pad200.bin | xor tiger.txt > tiger.bin
[cs342@puma] cat tiger.bin | showall
Q\225\314^F\317H^PdT\252\345\200\377+\334\337\230w^N^A4K\347sM\235X\327\310^H/\24
7^O\302^_*n\372EcB\323G$\213\263\225\230Q\316^O\364\235\326\310j\313zq^E^I^H^B\37
5\210.\206.\302\.^\313Y{\217,\376\360\207qUK^I\326y\254?p\325\230\357\252\372^I\
311D^\\327/\256\257c\376^B6^T\323\254e=<g^@\243\224^SuX\333\216e^Q\323\212\337yrJ
!}\254\202xu=C\245bgb<v=\2438:S^B\307\246&\331dg\366\237i
```

To see how this works, let's examine the operation on the first few characters of `tiger.txt` and `pad200.bin`:

File	**Chars**	**ASCII**	**Bits**
`tiger.txt`	`Tige`	`84 105 103 101`	`01010100 01101001 01100111 01100101`
`pad200.bin`	`^E\374\253c`	`5 252 171 99`	`00000101 11111100 10101011 01100011`
XOR of the two files	`Q\225\314^F`	`81 149 204 6`	`01010001 10010101 11001100 00000110`

The `xor` program generates a file whose size is that of the smaller of its two inputs. So the ciphtertext file `tiger.bin` has the same number of characters as the plaintext file `tiger.txt` (158). The `wc` program shows this:

```
[cs342@puma] wc tiger.bin
wc: tiger.bin:1: Invalid or incomplete multibyte or wide character
      0       4     158 tiger.bin
```

The `wc` program also indicates that `tiger.bin` has no newline character (`^J`), so it has 0 lines.

Of course, `tiger.bin` can be decoded by XORing it with the same pad that created it:

```
[cs342@puma] cat pad200.bin | xor tiger.bin
Tiger got to hunt,
Bird got to fly;
Man got to sit and wonder, "Why, why, why?"

Tiger got to sleep,
Bird got to land;
Man got to tell himself he understand.
```

`Tiger got to hunt,`
`We do, doodley do,`

`Bird got to fly;`
`doodley do, doodl`

`Man got to sit and wonder, "Why, why, why?"`
`ey do, What we must, muddily must, muddily m`

`Tiger got to sleep,`
`ust, muddily must, M`

`Bird got to land;`
`uddily do, muddily`

`Man got to tell himself he understand.`
`do, muddily do, muddily do, Until we b`

Figure 4: The interleaving of two poems, one in regular font and one in italics.

The `check` Program

I have create a program named `check` that helps to find text from two files that have been XORed. To motivate this program, consider another poem from *Cat's Cradle*:

```
[cs342@wampeter] cat doodley.txt
We do, doodley do, doodley do, doodley do,
What we must, muddily must, muddily must, muddily must,
Muddily do, muddily do, muddily do, muddily do,
Until we bust, bodily bust, bodily bust, bodily bust.
```

Suppose that we XOR the contents of `tiger.txt` and `doodley.txt`. The resulting file, which well call `mixed.bin`, looks like gibberish:

```
[cs342@puma] cat doodley.txt | xor tiger.txt > mixed.bin
[cs342@puma] cat mixed.bin | showall
^C^LG^A^]^LG^K^[O^P^CE^QU^J^[^@*&^F^]^@L^B^VTD^[C^@^B^C^V_f(^XND^HC~w^\^NTS^^^Q^@
^L^[^WT[O^C^Q^A^VEL[w^E^L_T[H^TYD^S^A^UF^Bg^?'^]KE^_U^C^K^]L^MOM^F^_^QIza^?&^M^[^
HYG^K^[^LT^BU^H^E^G^HB*)^NB^@^J^Z^PD^]^CYT^A^C@^@^E^\^I^W^L^@^_^@^L^J^L^?;^J^Q^[^
_T^V^KDL^?
```

How could we ever hope to decipher this? Observe that the `mixed.bin` is the result of XORing corresponding letters from the two poems, as suggested by the interleaving in Fig. 4. Note that the word `muddily` appears several times in the italic poem. If we moved a "sliding window" of this word over `mixed.bin` and XORed it at every position, then in the spots where `muddily` appear in `doodley.txt` we we see the corresponding characters of `tiger.txt`.

This is exactly what the `check` program does, as shown in the example in Appendix C. The input file (in this case, `mixed.bin`) is broken up into lines of 32 characters (in this case, five of them), and a sliding window of the given text (in this case, `muddily`) is "slid" over each line and XORed with its contents. Characters in the lines that do not participate in the XORing process are shown as asterisks. Because there are 32 characters in each of five lines, the result is 32 blocks with five lines each.

Most of the non-asterisk positions contain gibberish. But the spots where `muddily` appears in `doodley.txt` do *not* contain gibberish. Here are the relevant lines culled from Appendix C:

```
*******y, why?******************

*************************nder, "

***^L
Bird **********************

***************o land;**********

*********************r got t****

*******himself****************
```

Using this information, we know something about the contents of `tiger.txt`.

Of course, in practice, we're unlikely to pick `muddily` as a word to test for. It would make more sense to look for common words like `the`, `and`, `that`, `which`, and so on. We must also be careful with the results of checking such words, since "uncovered" words could come from *either* of the two source texts. Nevertheless, using trial and error, `check` can be used to uncover many words from the original texts. If a source text is searchable via Google, then sometimes a few choice words can help decode the whole text!

Appendix C: Example of the `check` Program

This is an example of Benny's `check` program in action. Six "interesting" lines have been hand-marked with `<==`.

```
[cs342@puma] cat mixed.bin | check muddily

ny#et‘>*************************
nc;^BAt7*************************
^Zph;=71*************************
|<^^^E^VJt*************************
n,0ej,y***********************

*a2eye+r************************
*{*^BLq"=************************
*hy;02$m************************
*$^O^E^[Oab************************
*4!eg)l|**********************

***tyh.gb***********************
**2^SL|’(q***********************
**a*0?!x ***********************
**^W^T^[Bdwq***********************
**9tg$iie*********************

***lhh#bw6**********************
***^K]|*-d:**********************
***2!?,}5=**********************
***^L
Bird **********************          <==
***lv$dlpp********************
```

```
****py#or#i********************
****Em* a/^G********************
****9.,p0(j********************
****^RSi^?a5>********************
****n5dauen******************

*****a2o^?&|z*******************
*****u; l*^R^N*******************
*****6=p=-^?x*******************
*****Kx^?l0+r*******************
*****-uax‘{u*****************

*******^^?+yo<******************
******#1l’^W^[e******************
******%a= zml******************
******‘nl=.gb******************
******mpxm^‘y****************

*******fn+tj)h*****************
*******)}’^Z^^pw*****************
*******y, why?*****************     <==
*******v}=#bwu*****************
*******himself***************       <==

********v:tg,},****************
********e6^Z^Sub-****************
********41we|*{****************
********e,#or‘-****************
********q|shisy**************

*********"eg!x9s***************
*********.^K^Sxg8****************
*********)feq/n^^***************
*********42o^?e8{***************
*********dbhdvlu*************

**********}v!u<fb**************
**********^S^Bxj=?g**************
**********^tq"k^K^F**************
***********^^?h=n,**************
**********zyd{i‘s************

***********n0u1cwy*************
***********^Zij0:rh*************
***********l‘"f^N^S^*************
***********fnh0k9q*************
***********au{defu***********

************(d1nrlS************
************q{07w}y************
************x3f^C^VKd************
************vy0f<d|************
************mjdhc‘^F**********

*************| n^??iF_***********
*************c!7zxlu***********
*************+w^C^[Nq2***********
*************a!f1ai^***********
```

```
*************ruhne^SB**********

**************8^?^?dCJ^?***********
**************9&zui‘b***********
**************o^R^[Ct’<***********
**************9w1llkq***********
**************mynh^VWs*********

***************gndNOjd**********
***************>kudewn**********
***************

Cy")f**********
***************o land;**********     <==
***************a^?h^[Rfh********

****************vuNBoqy*********
****************sddhr{-*********
****************^RRy/,s,*********
****************8}aca.S*********
****************gy^[_c}b*******

*****************m_Bbtl5********
*****************|uh^?^8"********
*****************Jh/!v9z********
*****************epcl+FP********
*****************a
_nxwf******

******************GSbyi {*******
******************my^?s=76*******
******************p>!{<or*******
******************hrl&CEw*******
******************^RNnurs-*****

*******************Ksyd%no******
*******************ans02#z******
*******************&0{1jgd******
*******************j}&N@b;******
*******************V^?u^?v8o****

********************khd(kz-*****
********************vb0?&oh*****
********************(j1gbq5*****
********************e7NMg.y*****
********************gd^?{=zr***

*********************pu(f^?8=****
*********************z!?+j}x****
*********************r got t****
*********************/_Mj+ls****
*********************|n{0^?g=**

**********************m9fr=(b***
**********************9.+gxmo***
**********************8voy%a6***
**********************G\j&ifc***
**********************vj0rb(5*
***********************!wr0-w:**
***********************6:guhz<**
***********************n^y(d#4**
***********************D{&dcvi**
***********************r!ro- ^F

************************oc0 r/y*
************************"vue^?)5*
************************fh(i&!^?*
************************c7dns|=*
************************9co %^S

*************************{! ^?*l{
*************************nder, "     <==
*************************p9i+$jf
*************************/un^y(d
*************************{^ (^V

z*************************91^?'in
z*************************|tr!%7
^N*************************!x+)os
h*************************m^?^t-q
z*************************f1(^[

ou*************************)n'dk
oo*************************lc!(2
^[|*************************‘:)bv
}0*************************got t     <==
o *************************)9^[

j‘>*************************v6df
jz&*************************{0(?
^^iu*************************"8b{
x%^C*************************we y
j5-*************************!


ge+x*************************.uf
g^?3^_*************************(9?
^Sl‘&************************* s{
u ^V^X*************************}1y
g08x*************************^R

gh.md*************************mw
gr6
Q*************************!.
^Sae3-*************************kj
^F*************************)h
g==mz*************************

vh#hqu*************************o
vr;^ODa*************************6
^Bah68"*************************r
d-^^^H^S_*************************p
v=0ho9************************
```