YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CPSC 467b: Cryptography and Computer Security *Handout #2*
Professor M. J. Fischer *January 20, 2010*

Problem Set 1

Due by midnight on Wednesday, February 3, 2010.

1 Goal

The goal of this problem is to show the feasibility of fully automating a brute-force attack on the Caesar cipher and to explore conditions under which this attack is effective. A brute force attack on a cryptosystem means trying all possible keys to see which one "unlocks" the encrypted message.

The difficulty in automating such an attack, aside from the time it takes to explore a potentially large key space, is knowing when you have succeeded. How can one distinguish the correct decryption from the wrong one? In general one cannot, but if some messages are more likely than others to be the decryption of a given ciphertext string c, then it is sensible to guess the most likely message m' from among the possibilities. The guess m' might or might not be equal to the original message m. If it is, we say the attack *succeeds*; otherwise it *fails*. We are interested in exploring how well this attack works at cracking the code.

2 Method

We begin by defining a message distribution M over length-r messages. The attacker knows the encryption $c = E_k(m)$ for a random message m chosen according to M and a key k chosen according to the uniform distribution over $\mathcal{K}$. The attacker also knows the parameters that define M as explained below. The attacker outputs a message m' and either succeeds or fails depending on whether $m' = m$ or not.

Let p be a probability distribution over the alphabet $\Sigma = \{\mathsf{A}, \mathsf{B}, \ldots, \mathsf{Z}\}$. Thus, $p(\mathsf{A})$ is the probability of choosing letter A. The message distribution M is the one that arises from the process of independently picking r letters at random according to p. Let $m = a_1, \ldots, a_r$. Then

$$P[m] = \prod_{i=1}^{r} p(a_i).$$

Thus, if $r = 3$,

$$P[\mathsf{CAT}] = p(\mathsf{C}) \cdot p(\mathsf{A}) \cdot p(\mathsf{T}).$$

Let $\mathcal{M}$ be the set of r-letter messages, and let $\mathcal{M}_c = \{D_k(c) \mid k \in \mathcal{K}\}$, the set of messages that encrypt to c under some key. These are the candidate decryptions for c.

The Caesar cipher has the property that $E_k(m)$ and $D_k(c)$, regarded as functions of k for fixed m and c, are one-to-one, i.e., different keys encrypt the same message as different ciphertexts, so also different keys decrypt the same ciphertext to different messages.[1] Hence, the probability that c

[1]For the Caesar cipher, it can be shown that $\{\mathcal{M}_c \mid c \in \mathcal{C}\}$ is actually a partition of $\mathcal{M} = \mathcal{C}$, and the relation $\equiv$ defined by $m_1 \equiv m_2$ iff $\exists c (m_1 \in \mathcal{M}_c \wedge m_2 \in \mathcal{M}_c)$ is an equivalence relation.

is the ciphertext of a randomly-chosen message m is simply $P[c] = \sum_{m \in \mathcal{M}_c} P[m]$.[2] Moreover, the conditional probability that m is the message given fixed ciphertext c is

$$\frac{P[m]}{\sum_{m' \in \mathcal{M}_c} P[m']} \tag{1}$$

if $m \in \mathcal{M}_c$ and is 0 otherwise. Hence, if the attacker sees ciphertext c, the best guess for m is one that maximizes (1).

3 Assignment

The assignment consists of three parts:

1. To write a C or C++ program to implement the above attack on a Caesar cipher.
2. To determine experimentally the success probability of this attack when applied to the message distribution M defined above and the attacker knows the underlying letter distribution p.
3. To determine experimentally how well the this attack works when applied to unknown English texts when the attacker uses a letter distribution derived from a fixed known text.

4 Programming notes

4.1 Needed functions

You should write a function `guess()` that finds the message $m' \in \mathcal{M}_c$ with highest probability and the key k for which $E_k(m) = c$. Ties are broken by choosing the message corresponding to the smaller key. Parameters to the function consist of a probability distribution p over the alphabet Σ, a message length r, and a suitably represented ciphertext of length r.

You will also need a function to read a frequency table from a file and convert it to a probability distribution for use with `guess()`. A frequency file consists of 26 whitespace-delimited integers, where the i^{th} integer gives the frequency of occurrence of the i^{th} letter of the alphabet. To convert these frequencies to probabilities, you will need to normalize by dividing each frequency by the sum of the frequencies.

5 Experiments

Your first job in this assignment is to measure experimentally the success probability of `guess()` under various conditions. First will be to try it on random messages using different letter distributions to see how the success probability is affected by distribution and message length. Second will be to try it on sample English-language sentences using two word-frequency tables derived from famous English texts: *The Merry Wives of Windsor* by William Shakespeare, and *Ulysses* by James Joyce.

[2] Here, we're regarding both m and c as random variables.

5.1 Experimental evaluation with random messages

An *experiment* takes two inputs: A probability distribution p on single letters and a length r. The steps of conducting an experiment are:

1. Choose a random message m of length r according to the distribution on length-r strings induced by p.
2. Choose a key k uniformly at random from $\{0, \ldots, 25\}$.
3. Compute $c = E_k(m)$, where E is the Caesar encryption function on length-r strings.
4. Run the procedure `kk = guess(p, c, r, mm)`.
5. If $\mathtt{mm} = m$, output *success* and the pair $(\mathtt{mm}, \mathtt{kk})$. Otherwise, output *failure*.

Finally, you should gather data by running a series of experiments. For each $r = 1, 2, \ldots$ up to some reasonable number (say 100), run a large number of experiments (say 100) and compute the fraction of successes. Repeat this several times to get a feeling for the variance in your results. Plot the results and find numbers r_1 and r_2 (if possible) such that the observed success rate is less than 10% for $r < r_1$, between 10% and 90% for $r_1 \leq r \leq r_2$, and greater than 90% for $r > r_2$. You will find some frequency files on the Zoo in /c/cs467/assignments/ps1 to run your experiments on. You should run your experiments on each distribution to see the effects of the distribution on the success probabilities. See the Readme file for detailed descriptions of what these distributions are and how they were obtained.

5.2 Experimental evaluation with English texts

You will also find some sample encrypted English texts. You should run your algorithm on each of them, using both the “”merry wives” and the “ulysses” frequency tables and report whether or not it successfully decrypts them.

6 Further programming and submission details

Beyond this description, you are free to organize your code any way you see fit. Keep in mind that your code has multiple purposes. Beside showing me (and the TA) that you know how to program in C/C++, the purpose is to get an understanding through experimental means of how well a simple cryptanalysis tool can do under controlled circumstances and to see how its efficacy depends on the assumed underlying probability distribution on the message space and on the length of the message. Designing experiments and presenting experimental results in a compelling way is a useful skill to acquire, quite independent of the particular topics of cryptography and security on which this course focuses.

One caveat: Experimental results are worthless if the experimental code is flawed, so you also need to make a convincing case that your code is correct, i.e., correctly implements the scheme described by the problem assignment. Programs that generate random numbers can often be hard to debug because runs may not be repeatable. To make them repeatable, you should explicitly seed your random number generator during debugging. Of course, for production runs where you are performing 100 trials, each trial will need its own random seed. The result of `time(NULL)` can be used for this purpose since it will necessarily be different on each run. There are many random number generators available. For this assignment, you should use the standard `rand()` and `srand()` functions.

If you write your program in the straightforward way, you will likely run into floating point underflow problems when computing the probability of long messages. The easiest solution to this problem is to compute the logarithm of the probability rather than the probability itself. This works because you are not really interested in the actual value of the probability but only in determining which of the probabilities is the largest. Since the logarithm function is strictly monotonic increasing, the largest probability will also have the largest logarithm. To compute the logarithm of a product pq, use the formula $\log(pq) = \log(p) + \log(q)$. The base of the logarithms you use does not matter as long as you use the same base for all of your calculations.

When your program is working, you should gather data by running the experiments described above on the designated probability distributions. The results of your experiments should be presented in both tabular and graphical form. You may use any tools you wish to prepare your report. In particular, it is perfectly acceptable to use the graphing tools in spreadsheets to display your data or to use any other available graphing package.

You should submit your work using the submission script in the `/c/cs467/bin` course directory on the Zoo. Your submission should include the program or programs you have written, test runs showing the correctness of the various pieces, and the data tables and graphs resulting from your experiments. You should also submit written documentation describing in some detail what you have done.