Massachusetts Institute of Technology
6.857: Network and Computer Security
Professor Ronald L. Rivest

Handout 10
September 30, 2003

Problem Set 2 Solutions

Today's fun-fact is about jointly-authored submissions. During your professional lives, you will frequently be asked to jointly submit a letter, a paper, or an article. When you submit a document that has multiple names on it, *you should not write sentences that use the word "I" as their subject.* Doing so is a dead give-away that the paper was not authored by the group, but was instead authored by a single person. (It also comes across as somewhat haughty to the reader.)

For example, this sentence appeared in a homework solution to problem set #2 that had four names on the top:

> The approach I used to solve this problem is as follows.

How should we take such a sentence?

Please remember that these homeworks are meant to be done in groups. This means that each student is responsible for all of the work. It does not mean that you have license to split up the problems and have each student work on a different one.

Submitted homework solutions that used the first-person pronoun were penalized between 1 and 2 points.

Problem 2-1. Two-time pad (Courtesy of Rui L. Viana, Conor M. Murray, Pallavi Naresh and Richard Hansen.)

TA Notes: Many students enjoyed the two-time pad problem, and so many people got it that we should make the problem harder next year. The vast majority of students tried XORing each of the texts with the other texts and then seeing which of the 3 results had no high-order bits set. One group of students did a statistical analysis of the distribution of the resultant XOR strings with the theoretical distribution of an XOR of standard english text against standard english text, but it turned out that this approach was not needed.

Once the pairs were distinguished, most students wrote a program that allowed them to type in sample text and then to see the results of XORing the text in a 1-character sliding window along the entire length of the chosen combined text. This seems to have been a pretty manual process, although one group of students automated the process by doing a brute-force dictionary lookup off all the words in the `web2a` file and then doing a statistical analysis on the result. (As it turns out, it was easier to work manually because of the high predictability of the text.) One of those most interesting experiences in solving this assignment is reported below:

> Oddly enough, we had the movie "Cube" playing, and got to talking about people dying. Some how we someone suggested Johnny Cash, and we tried "Ring of fire." Oddly enough our program spouted (among many other things) a bunch of primes. (This really freaked us out, because as I said ... we had just been watching Cube.)

Our only dissappointment with the example was that many students, once they identified the source string, simply reported the source string, rather than *what was actually in the plaintext.* As a result, many students reported that the Hamlet quote ended with the word "many a day" rather than "many a" or "many a d". (In fact, the quote ended with a single 'd' followed by a newline.) Likewise, the set of prime numbers should have been reported as ending with a single digit "1," as the following 3-digit number was similarly truncated. Depending on how egregious the misreporting was, between 0 and 2 points were taken off.

The following solution was submitted by Rui L. Viana, Conor M. Murray, Pallavi Naresh and Richard Hansen:

(a) *Approach* Our approach was to first analyze the messages to find the pairs and then to write a program to assist us in the guessing process. The properties of exclusive or (XOR) aided us in finding the pairs and helped us gain information about the plaintext (see the section on decrypting below).

We exploited the predictability of language by trying common words (both English and code) in search of likely plaintext. Most of the trial and error was not automated as it required a user to analyze the results of one guess, and decide the next course of action effectively. Though Rui wrote code to run the messages through a dictionary in search of matches, which may have helped in larger problems involving two-time pads, it proved unnecessary.

Finding the Pairs If we assume the plaintexts are in (non-extended) ASCII code then we know the first bit of each byte is zero. For two cipher texts encrypted with the same key, the leading 0's on every byte will be encrypted to the same bit, hence when we XOR two messages that used the same pad we would expect the first bit of every byte to be 0. We wrote a program to print out the XOR value of two input files in hex (a leading 0 in binary corresponds to 7 or less in hex) and inspected the values. By trial and error we were able to find exactly three pairs of files that fit this criterion and so were we confident these pairs were encrypted using the same pad and that the files were encoded in ASCII.

The assumptions on which we based our matching of the files could have been incorrect. We were however able to make progress using these pairs in a reasonable amount of time. If working with these pairs had not been fruitful, further analysis would have been required, perhaps eliminating the ASCII encoding assumption. In this case, however, it was unnecessary.

Decrypting We wrote a program that took, as input, 2 files (C_1 and C_2) and a string (S). The output would be the plaintext of one of the messages corresponding to S i.e. we would enter a guess at a substring of one of the plaintexts and the program would output the plaintext of the other message given this string. It would output $(n - b + 1)$ strings of the same length as S corresponding to all the possible positions of the substring. This works because of the properties of XOR:

$$C_1 = M_1 \oplus K, \quad C_2 = M_2 \oplus K$$
$$C_1 \oplus C_2 = (M_1 \oplus K) \oplus (M_2 \oplus K) = M_1 \oplus M_2$$

(From 6.857 Lecture 4 notes)

Now we had an efficient way to guess at substrings and view the results. We started guessing common words (e.g. "The ", " the "). We discovered the beginning of C code in messages 1 and 3 and were able to guess at the code until recovering enough of the play script to perform a Google search for it.

To decrypt messages 4 and 5 we guessed common English words until it seemed like the corresponding message could be in English, usually swapping from one message to the other as we were able to expand the strings. It was speeded up by recognizing the TA's names and text from the problem set.

The last pair (2 and 6) was the hardest to decrypt. By looking at the first byte in more detail we determined that if in one message it was a letter in the other it must be a number. We knew the second bytes in the messages were equal (the second byte of $M_2 \oplus M_6$ was `0x00`) and we guessed it to be a space. This led us to think about sequences of numbers, soon after arriving at the primes.

Here is the Java code for the program we used to assist us in decoding the ciphertexts (Decrypt.java):

```
import java.io.FileInputStream;

public class Decrypt {
  public static void main(String[] args) throws Exception {
    if(args.length!=3) {
      usage();
      System.exit(0);
    }
    xor(args[0],args[1],args[2]);
  }

  public static void usage() {
    System.out.println("Usage:");
    System.out.println("java Decrypt cipher1 cihper2 word");
  }


  /* prints the result of cipher1 (xor) cipher2 (xor) theWord, where
  theWord is taken to start at every possible location.
  @param cipher1FileName the name of the first cipher text
  @param cipher2FileName the name of the second cipher text
  @param theWord the word to be matched against the xor
  */
  public static void xor(String cipher1FileName,
  String cipher2FileName,
  String theWord) throws Exception {
    // the length of the cipher texts
    int length = 128;
    // open the two cipher text files
    FileInputStream file1 = new FileInputStream(cipher1FileName);
    FileInputStream file2 = new FileInputStream(cipher2FileName);

    // f1 will contain the bytes of cipher1 and f2 the bytes of
    // cipher2.
    int[] f1 = new int[length];
    int[] f2 = new int[length];
    // xor will contain the xor of the two ciphers, i.e., the xor of
    // the two plain texts.
    int[] xor = new int[length];

    // intialize arrays
    for(int i=0;i<length;i++) {
      f1[i]=0; f2[i]=0; xor[i]=0;
    }

    // xor the cipher texts
    for(int i=0;i<length;i++) {
      f1[i] = file1.read();
      f2[i] = file2.read();
      xor[i]= f1[i]^f2[i];
      if(f1[i]==(-1) && f2[i]==(-1)) break;
    }

    // wordLength is the length of the word to be compared against the
    // xor and the word[] is the same as theWord except that it is
    // an array.
    int wordLength = theWord.length();
    int[] word = new int[wordLength];
    // construct word[]
    for(int i=0;i<wordLength;i++) {
      word[i] = (int)theWord.charAt(i);
    }

    // now we'll xor word[] against the xor os the msgs, starting at
    // each byte.
    for(int i=0;i<length-(wordLength-1);i++) {
      System.out.print(i+" -------> ");
      // possible contains the result of xor between word[] and xor[]
      // starting at byte i
      char[] possible = new char[wordLength];
      // do the xor and print at the same time
      for(int j=0;j<wordLength;j++) {
        possible[j] = (char)(word[j]^xor[i+j]);
        // only print character if it's nothing weird
        if(possible[j]>31 && possible[j]<127) {
          System.out.print(possible[j]);
        }
        // if it is weird, print the int value of the character
        else System.out.print("{"+((int)possible[j])+"}");
      }
      System.out.println("");
    }
  }
}
```

(b) *Pairs*

- Messages 1 and 3
- Messages 2 and 6
- Messages 4 and 5

(c) *Plaintexts*

- Message 1 (or 3):

```
========================================================================
Offset              Message Bytes (in hex)                     ASCII
========================================================================
  00   54 68 65 20 66 61 69 72 20 4F 70 68 65 6C 69 61  The fair Ophelia
  10   21 20 4E 79 6D 70 68 2C 20 69 6E 20 74 68 79 20  ! Nymph, in thy
  20   6F 72 69 73 6F 6E 73 20 0A 42 65 20 61 6C 6C 20  orisons  Be all
  30   6D 79 20 73 69 6E 73 20 72 65 6D 65 6D 62 65 72  my sins remember
  40   27 64 2E 0A 4F 50 48 45 4C 49 41 0A 47 6F 6F 64  ’d. OPHELIA Good
  50   20 6D 79 20 6C 6F 72 64 2C 0A 48 6F 77 20 64 6F   my lord, How do
  60   65 73 20 79 6F 75 72 20 68 6F 6E 6F 75 72 20 66  es your honour f
  70   6F 72 20 74 68 69 73 20 6D 61 6E 79 20 61 20 0A  or this many a
========================================================================
```

- Message 2 (or 6):

```
========================================================================
Offset              Message Bytes (in hex)                     ASCII
========================================================================
  00   32 20 33 20 35 20 37 20 31 31 20 31 33 20 31 37  2 3 5 7 11 13 17
  10   20 31 39 20 32 33 20 32 39 20 33 31 20 33 37 20   19 23 29 31 37
  20   34 31 20 34 33 20 34 37 20 35 33 20 35 39 20 36  41 43 47 53 59 6
  30   31 20 36 37 20 37 31 20 37 33 20 37 39 20 38 33  1 67 71 73 79 83
  40   20 38 39 20 39 37 20 31 30 31 20 31 30 33 20 31   89 97 101 103 1
  50   30 37 20 31 30 39 20 31 31 33 20 31 32 37 20 31  07 109 113 127 1
  60   33 31 20 31 33 37 20 31 33 39 20 31 34 39 20 31  31 137 139 149 1
  70   35 31 20 31 35 37 20 31 36 33 20 31 36 37 20 0A  51 157 163 167
========================================================================
```

- Message 3 (or 1):

```
========================================================================
Offset              Message Bytes (in hex)                     ASCII
========================================================================
  00   23 69 6E 63 6C 75 64 65 20 3C 73 74 64 69 6F 2E  #include <stdio.
  10   68 3E 0A 23 69 6E 63 6C 75 64 65 20 3C 73 74 64  h> #include <std
  20   6C 69 62 2E 68 3E 0A 23 69 6E 63 6C 75 64 65 20  lib.h> #include
  30   3C 73 74 72 69 6E 67 2E 68 3E 0A 23 69 6E 63 6C  <string.h> #incl
  40   75 64 65 20 3C 65 72 72 2E 68 3E 0A 0A 6D 61 69  ude <err.h>  mai
  50   6E 28 69 6E 74 20 61 72 67 63 2C 20 63 68 61 72  n(int argc, char
  60   20 2A 2A 61 72 67 76 29 0A 7B 0A 20 20 20 20 69   **argv) {     i
  70   66 28 61 72 67 63 21 3D 33 29 7B 0A 09 66 70 0A  f(argc!=3){  fp
========================================================================
```

- Message 4 (or 5):

```
========================================================================
Offset              Message Bytes (in hex)                     ASCII
========================================================================
  00   53 6C 6F 77 6C 79 2C 20 79 6F 75 20 72 65 61 6C  Slowly, you real
  10   69 7A 65 20 74 68 61 74 20 61 20 6F 6E 65 2D 74  ize that a one-t
  20   69 6D 65 20 70 61 64 20 63 61 6E 20 6F 6E 6C 79  ime pad can only
  30   20 62 65 20 75 73 65 64 20 6F 6E 63 65 2E 20 59   be used once. Y
  40   6F 75 20 68 61 76 65 20 73 69 78 20 6D 65 73 73  ou have six mess
  50   61 67 65 73 2D 2D 2D 73 6F 20 73 6F 6D 65 20 6F  ages---so some o
  60   66 20 74 68 65 20 70 61 64 73 20 6D 75 73 74 20  f the pads must
  70   68 61 76 65 20 62 65 65 6E 20 72 65 2D 75 73 0A  have been re-us
========================================================================
```

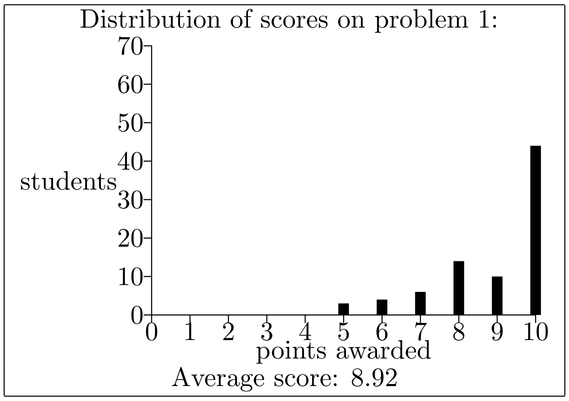
- Message 5 (or 4):

```
========================================================================
Offset              Message Bytes (in hex)                     ASCII
========================================================================
  00   53 69 6D 73 6F 6E 20 47 61 72 66 69 6E 6B 65 6C  Simson Garfinkel
  10   20 61 6E 64 20 43 68 72 69 73 74 6F 70 68 65 72   and Christopher
  20   20 50 65 69 6B 65 72 74 20 54 41 20 36 2E 38 35   Peikert TA 6.85
  30   37 20 69 6E 20 36 2D 31 32 30 2E 0A 54 68 69 73  7 in 6-120. This
  40   20 6D 65 73 73 61 67 65 20 69 73 20 76 65 72 79   message is very
  50   20 64 69 66 66 69 63 75 6C 74 20 74 6F 20 64 65   difficult to de
  60   63 72 79 70 74 2C 20 62 75 74 20 79 6F 75 20 63  crypt, but you c
  70   61 6E 20 64 65 63 72 79 70 74 20 69 74 2E 0A 0A  an decrypt it.
========================================================================
```

- Message 6 (or 2):

```
========================================================================
Offset              Message Bytes (in hex)                     ASCII
========================================================================
  00   49 20 66 65 6C 6C 20 69 6E 74 6F 20 61 20 62 75  I fell into a bu
  10   72 6E 69 6E 67 20 72 69 6E 67 20 6F 66 20 66 69  rning ring of fi
  20   72 65 0A 49 20 77 65 6E 74 20 64 6F 77 6E 20 64  re I went down d
  30   6F 77 6E 20 64 6F 77 6E 0A 41 6E 64 20 74 68 65  own down And the
  40   20 66 6C 61 6D 65 73 20 77 65 6E 74 20 68 69 67   flames went hig
```

```
 50   68 65 72 0A 41 6E 64 20 69 74 20 62 75 72 6E 73  her And it burns
 60   20 62 75 72 6E 73 20 62 75 72 6E 73 0A 54 68 65   burns burns The
 70   20 72 69 6E 67 20 6F 66 20 66 69 72 65 0A 54 0A   ring of fire T
=================================================================
```



Problem 2-2. Hash Soup (Courtesy of Rui L. Viana, Conor M. Murray, Pallavi Naresh and Richard Hansen.)

TA Notes: There were several common mistakes made in this problem: not explicitly stating assumptions (especially about MD5 and SHA-1), making a mathematical error and not "gut-checking" the final answer, misunderstanding the "birthday paradox" nature of the problem, and not settling upon a *specific* non-trivial probability (as opposed to an algebraic expression) for the number of files in the world or probability of collision.

In order to do this problem, one must make a specific assumption about the hash functions in question. One might assume that someone has completely broken MD5; in this case, there is *certainly* a pair of non-identical files having the same hash code! (Nobody assumed this, though.) Alternatively, one should assume that MD5 and SHA-1 are pseudorandom, i.e. it is infeasible to correlate their outputs on different inputs. (Remember: these hash functions are not *proven* to be pseudorandom, it's just a widely-believed assumption!)

Many groups made some kind of mental or mathematical error, such as confusing the probability that there *exists* a collision with the probability that there is *no* collision. This led to claims that there is *certainly* a pair of non-identical files which collide, when in fact the probability of such an event is extremely small (about 2^{-40}, or one in a quadrillion, for MD5 under reasonable assumptions).

About 5 points were allocated for a reasonable estimate of the number of unique files on the planet, and 5 points for a correct mathematical analysis that settled upon a specific, non-trivial probability.

The following solution was submitted by Rui L. Viana, Conor M. Murray, Pallavi Naresh and Richard Hansen:

This problem asks us to estimate the probability that there are two non-identical files in the world that have the same hash code. We believe that this is not a very useful exercise for two reasons: (1) It is difficult to estimate the number of files in the world within a few orders of magnitude so any probability estimate will have little useful value, and (2) few people really care about the actual probability—they just want to know if a collision is improbable enough to not be a security concern.

So, instead of trying to determine a solution to the problem as asked, we are going to answer what we assume to be the underlying question: obtain upper bounds on the probabilities.

This question is much like that posed by the classic birthday problem: given a group of size m, what is the probability that at least two members of the group have the same birthday. We can generalize this problem as follows: given q balls and N bins, if we randomly assign balls to bins, what is the probability that at least two balls will fall in the same bin?

In our case, the balls are the set of all files in the world and the bins are all the possible output hashes resulting from each respective hashing algorithm.

To calculate the total number of files in the world q, we can make the following estimate:

According to some estimates there are approximately half a billion installed PCs in the world [1]. Let us make this estimate more generous, and assume there are a total of 1 billion PCs and file servers in the world. Let us be even more generous and assume that on average, each machine contains approximately 1 million unique files. This gives us a total of 1×10^{15} files in existence.

[This is an extremely generous estimate. Most home PCs have less than 100,000 files, most of which are not unique. The number of file servers is most certainly no where near the number of PCs and even file servers do not contain 1 million unique files on average.]

From the solution to the generalized birthday problem, we can calculate the probability that two files hash to the same value as follows (from [2]):

Let C denote the event that at least two files hash to the same value. Let N denote the number of (equiprobable) hash values. Let q denote the number of files in the world.

$$\Pr(C) = 1 - \frac{N!}{(N-q)!N^q}$$

From [3], the lower and upper bounds on this equation are:

$$\Pr(C) \leq \frac{q(q-1)}{2N}$$

$$\Pr(C) \geq 1 - e^{-q(q-1)/2N}$$

$$\Pr(C) \geq 0.3\frac{q(q-1)}{N}$$

For MD4, $N = 2^{128} \approx 3 \times 10^{38}$. Therefore,

$$\Pr(C_{\mathrm{MD4}}) \lessapprox \frac{1 \times 10^{30}}{7 \times 10^{38}} \approx 1 \times 10^{-9}$$

For SHA-1, $N = 2^{160} \approx 1 \times 10^{48}$.

$$\Pr(C_{\mathrm{SHA-1}}) \lessapprox \frac{1 \times 10^{30}}{3 \times 10^{48}} \approx 3 \times 10^{-19}$$

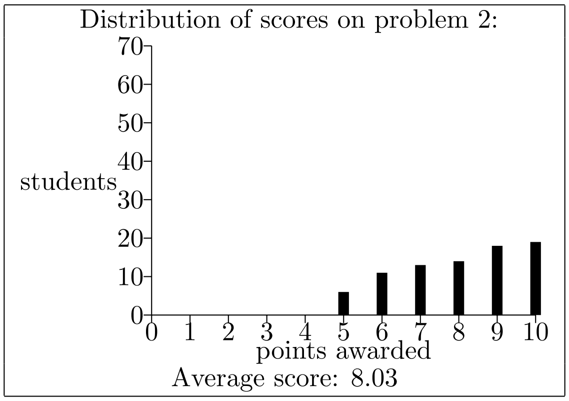
As we can see, for both MD4 and SHA-1, the size of N is so much greater than q that the probability of a collision somewhere in the world is quite remote (even with our extremely generous estimates).

References

[1] `http://www.pipe21.com/researchportal/_samples/reports/sample1.htm`

[2] `http://mathworld.wolfram.com/BirthdayProblem.html`

[3] `http://www.cs.ucsd.edu/users/mihir/papers/gb.pdf`, page 242



Problem 2-3. One-Time MAC, Revisited (Courtesy of Rohit Rao, Joy Forsythe, Leah Oats, and Sharon Cohen.)

TA Notes: The biggest mistake on this problem was arguing that, in order to make a forgery, the adversary must guess *the entire key* $(a_1, \ldots, a_t, b)$. In fact, given a valid pair (M, T), and for a fixed choice of message M' and tag T', there are *several* keys $\tilde{K}$ for which $T' = \mathrm{MAC}_{\tilde{K}}(M')$ and $T = \mathrm{MAC}_{\tilde{K}}(M)$ (there are p^{t-1} such keys, to be precise). There's no longer a one-to-one mapping between potential keys and the corresponding MAC tags for M'. With this in mind, an argument like the one given in lecture doesn't go through without some modifications.

One geometric way to interpret this new MAC is in $t+1$ dimensions (the special case of $t = 1$ is exactly the MAC we saw in lecture). The t free dimensions correspond to the message blocks $M_1, \ldots, M_t$, and the $(t+1)$st dimension corresponds to the MAC value for the given blocks. The key specifies a t-dimensional hyperplane on which all the valid (message, MAC) pairs lie. The adversary is given one point (M, T) on the plane, which yields no information about the orientation of the plane. For every message $M' \neq M$, every hypothesized MAC tag T' is equally likely (there are exactly p^{t-1} planes that go through both (M, T) and (M', T')). Therefore the best the adversary can do is guess the correct tag with probability $1/p$.

The following solution was submitted by Rohit Rao, Joy Forsythe, Leah Oats, and Sharon Cohen:

Proof Idea: The general idea of this proof follows directly from the proof of the one-time MAC that was given in class. We know that $\mathrm{MAC}_k(M) = (\sum a_i M_i) + b$. We think of this new one-time MAC as a series of traditional one-time MAC operations:

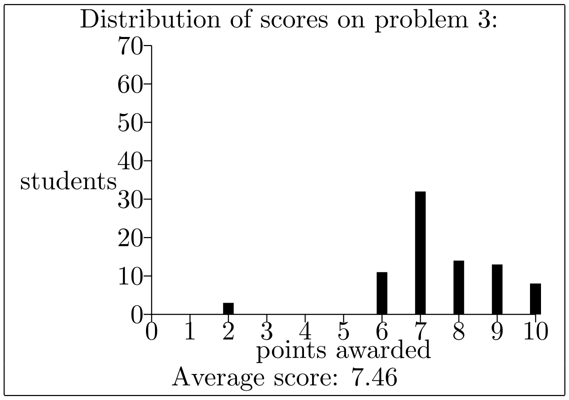
$$\begin{aligned}
Y_0 &= b \pmod{p} \\
Y_1 &= a_1 M_1 + Y_0 \pmod{p} \\
Y_2 &= a_2 M_2 + Y_1 \pmod{p} \\
&\cdots \\
Y_t &= a_t M_t + Y_{t-1} \pmod{p}
\end{aligned}$$

By recursively substituting in for the Ys and gathering terms, we can see that Y_t is the one-time MAC described in the problem set.

To show that this MAC is secure, let's look at it inductively. Suppose the adversary is given a valid $(M, \mathrm{MAC}_k(M))$ pair, and is trying to forge the MAC for a message M'. We know $Y'_0 = b$ is uniformly random (even conditioned on what the adversary knows). Now suppose $M'_i \neq M_i$; then $Y'_i = a_i M'_i + Y'_{i-1} \pmod{p}$ is random and independent of the adversary's view, due to the security of the one-time MAC presented in class. Inductive case: given that Y'_i is uniformly random (conditioned on the adversary's knowledge), then $Y'_{i+1} = a_{i+1} M'_{i+1} + Y'_i \pmod{p}$ is too, because the term Y'_i is. Therefore Y'_t is uniformly random, even against an unbounded adversary, and the MAC is secure.

For a message M of length n, we can compute the number of key bits necessary for both schemes. Old: $M \in Z_p \Rightarrow |p| \geq n$, and $|k| = 2|p| \Rightarrow |k| \geq 2n$. New: $M_i \in Z_p$ for all i implies $|p| \approx n/t$ where t is the number of message blocks. $|k| = (t+1)|p| \Rightarrow |k| \approx (t+1)n/t = n + n/t$. If we choose a fixed block size s, then we can let $t = n/s$. Then $|k| \approx n + s$.

For large n, the new scheme will nearly halve the number of key bits that need to be generated.


Distribution of scores on problem 3:
students
points awarded
Average score: 7.46