CSCI3381 Cryptography-Assignment 2
Perfect secrecy, pseudorandom number generators, stream ciphers

due Wednesday, February 9 at 11:59 PM

January 27, 2017

1 Overview

The first part of this assignment consists of some pencil-and-paper 'theory' questions, organized around the definition of perfect secrecy. Although we will leave perfect secrecy behind (because it is essentially unattainable in practice), the formal definition of perfect secrecy serves as a springboard to formulating a precise definition of practical security, and the exercises remain relevant.

The second part consists of programming problems in which you are asked to break some stream ciphers that were poorly implemented.

2 Problems about perfect secrecy.

1. Consider a monoalphabetic substitution cipher applied to plaintexts consisting of just a *single letter*. That is,

   $$\mathcal{M} = \{0, 1, \ldots, 25\}.$$

   The keyspace $\mathcal{K}$ is the set of all permutations of $\mathcal{M}$.

   (As usual, we identify the letter $A, \ldots, Z$ with this set of integers.) Show that this is perfectly secret.

2. Now let us modify the Caesar cipher so that once again,

   $$\mathcal{M} = \{0, \ldots, 25\},$$

   but

   $$\mathcal{K} = \{1, \ldots, 25\}.$$

The encryption function is the same:

$$E(k, m) = (k + m) \bmod 26.$$

Because we have eliminated $k = 0$ as a possibility, we do not allow a plaintext letter to be 'encrypted' by itself. Show that this system is *not* perfectly secret. Find two messages m_1, m_2 such that in the indistinguishability experiment performed with these two plaintexts, the adversary can distinguish an encryption of m_1 from an encryption of m_2 with probability greater than $\frac{1}{2}$.

3. *Exactly* the same problem as above, but with this difference: The key space is now
$$\mathcal{K} = \{0, 1, \ldots, 30\}.$$

4. Now consider a monoalphabetic substitution cipher where, as usual, $\mathcal{K}$ is the set of all permutations of $\{0, \ldots 25\}$, and $\mathcal{M}$ consists of all *two-letter* sequences. Show that this system is *not* perfectly secret. Find two messages m_1, m_2 such that in the indistinguishability experiment performed with these two plaintexts, the adversary can distinguish an encryption of m_1 from an encryption of m_2 with probability greater than $\frac{1}{2}$.

5. The purpose of this problem is to demonstrate how an insecure pseudorandom generator leads to an insecure encryption method. Here
$$\mathcal{M} = \{0, 1\}^{256},$$
but
$$\mathcal{K} = \{0, 1\}^{128}.$$
Encryption and decryption are given by
$$E(k, m) = D(k, m) = G(k) \oplus m,$$
where $G : \{0, 1\}^{128} \to \{0, 1\}^{256}$. That is, this is a stream cipher with pseudorandom generator G.

A design flaw is discovered in the PRG: It turns out that approximately $\frac{2}{3}$ of the keys k lead to $G(k)$ having least significant bit 1. Find two messages m_1, m_2 such that in the indistinguishability experiment performed with these two plaintexts, the adversary can distinguish an encryption of m_1 from an encryption of m_2 with probability greater than $\frac{1}{2}$. (NOTE: Since the keys are much shorter than the messages, we know that this cipher does not have perfect secrecy, but we do not *a priori* know if there is an efficient method for distinguishing which of two plaintexts is encrypted. You need to show that in this case there is a very simple method that is correct significantly more than half the time.)

3 Bad crypto in the wild 2: Misusing random number generators

Let us recall the basic definitions. A pseudorandom number generator (PRG) is an easily-computable function G takes a seed value $s \in \{0,1\}^k$, and produces a longer bit string $G(s) \in \{0,1\}^{\ell(k)}$ that 'looks' random. The idea is that for a small number k of truly random bits, you obtain a large number $|\ell(k)|$ of apparently random bits.

A stream cipher built on the PRG G uses the shared key s to encrypt and decrypt messages of length $\ell(|s|)$:

$$m \mapsto G(s) \oplus m.$$

Every programming environment has a built-in random number generator (such as the `random` package in Python), but these are not *cryptographically secure* random number generators and can never be used for cryptographic applications like stream ciphers. The second problem below illustrates this with an implementation of Java's built-in random number generator.

Even if the PRG is cryptographically secure, if there is not a sufficient amount of randomness built in to the seed that is used, the resulting stream cipher is insecure. This is illustrated in the first problem below. The problem uses the RC4 stream cipher. While there are weakness in RC4, for purposes of our problem, we can suppose that it is secure. The attack in the problem is based on poor generation of the key.

This problem is inspired by real-life failures: In the 1990's, Netscape, which first developed the SSL protocol for encrypting Internet communication, released a version in which the PRG was seeded by the time of day, the process ID, and the parent process ID. These quantities taken together did not provide sufficient randomness, and it was possible to predict the seed and break the security of the system. In 2008 it was discovered that a bug had inadvertently been introduced into the Debian Linux implementation of SSL, so that the number of available values used to seed the random number generator was reduced to 32,768. (You can read a bit about these attacks, and find links to more information, in the Wikipedia article 'Random number generator attack'.

6. An implementation of the RC4 stream cipher is provided on the course website. As you can see, both the algorithm for producing the initial state from the seed value, and for updating the state and computing the next output byte of the stream, are very simple.

(a) (In this part of the problem you are not launching an attack, just decrypting something.)

Because RC4 is a stream cipher, keys cannot be reused, and thus a new key must be generated and somehow shared with each new message. One way to address this problem is the following scheme: Alice and Bob share a long-term key k, which in our example will be 40 bits. To encrypt a new message m, the sender generates a random string (in this example, 24 bits) called an *initialization vector*, or IV. The IV is concatenated to the key and use to encrypt the message, and then the IV is concatenated again to the ciphertext, so that the recipient receives

$$IV||E(IV||k, m).$$

To decrypt, the recipient strips away the 3-byte IV at the start of the message, concatenates that to the shared key k, and then decrypts. (This is exactly how things are done in the WEP encryption standard for for WiFi networks. This rather primitive way of combining the IV and the key, coupled with subtle weaknesses in RC4, led to devastating attacks on WEP.)

Alice and Bob share the 40-bit key whose hex representation is

$$7f65a580cb$$

(I copied this off of a WiFi router!) She sends Bob the ciphertext whose base 64 representation is posted on the website. Decrypt the message.

(b) This part of the problem does not involve an initialization vector, but just seeds the RC4 generator with the time. The problem is that this does not supply enough randomness at the outset.

Alice wants to encrypt a text file on her disk with RC4. (She will need a different key for each file she encrypts, which is one reason for not using a stream cipher to encrypt files on disk.)

She seeds the RC4 generator using the following code:

```
import time
s=int(time.time())
k=str(s)
```

That is, she strips the fractional part from the time, and then converts the time, which is 10 decimal digits long, into the corresponding ASCII string of digits. That is, she is using a 10-byte=80-bit key, which looks pretty random to her.

She writes the value of the key in a safe place, and then uses it to RC4-encrypt her file, then converts it to base64. The result is posted on the website.

You steal Alice's laptop and attempt to decrypt the file. While you cannot read it at first you are still able to read the system information about the file and find that it was created on January 27, 2017. You are pretty sure that the original was a plain text file in normal English.

Recover the plaintext.

7. I have posted a Python implementation of the built-in random number generator in Java, together with code for using it as a stream cipher. It is a *linear congruential* generator, and is definitely not cryptographically secure. The generator maintains a 48-bit state. Each time `next_bytes()` is called, the state is updated via a simple formula, and the high-order 32 bits of the state are returned as the next block of four bytes in the pseudorandom stream. More precisely, if the current state is s, then the next state is given by

$$s' = (25214903917 \times s + 11) \bmod 2^{48}$$

and the next four bytes of output are $\lfloor s'/2^{16} \rfloor$. The state was initialized by supplying a secret pass-phrase to `init_state()`. Observe that the first four bytes of the output are the high-order 32 bits of the next state after the initial one. Your job is to guess the other 16 bits of the state and decrypt the ciphertext, which is posted in base64 format on the website. This time I will provide you with a little bit of known plaintext: the first four characters of the plaintext are `FROM`.

4 What to hand in

You will hand in a zipped folder FirstIntialLastNameHW2.zip. The folder will contain a text-processed file giving your written solutions, including the explanation of your solutions to the programming problems. It will also contain a Python file FirstInitialLastNameHW2.py containing all the code you wrote to solve problems 6 and 7. Part of the code should contain the two functions with names prescribed as below.

You should structure your solutions to 6 and 7 in the following way: The solution to 6(a) will contain a function `rc4_iv_decrypt(key,ciphertext)` that takes as input the shared key, and a ciphertext with a 24-bit IV prepended, and returns the plaintext that was encrypted with a combination of the IV and ky. You can call the RC4 code that was supplied. (This problem is very easy, once you understand what it is saying.)

For Problem 6(b), supply a function `find_key6(ct)` that searches for the correct key value to decrypt the ciphertext `ct` and returns it as a result. So the return value will be a string like `'1485546517'` and not the recovered plaintext. You will of course use this to recover the plaintext, so there will be more functions, which we won't test for grading.

The solution to Problem 7 will likewise contain a function `find_key7(ct)` that returns the *hex* representation of the low order 16 bits of the state immediately after the initial state of the generator. Once again, there will be more code that uses this value to obtain the decryption of the ciphertext, but this is the function that we will test. In your writeups for 6 and 7, you will give the recovered plaintext, and explain how your solutions work–that is, what is the strategy that you followed to find the key: how many values did you need to test, how could you determine if the guessed value was the correct one?