Assignment 2

CSCI3381-Cryptography

Due September 24, 2014

These problems concern the one-time pad, perfect secrecy, and stream ciphers. For the programming problems, ASCII plaintexts and base 64-encoded ciphertexts are posted on the website; you should be able to copy them and paste them into your Python code. Note that for the last problem, which is the most involved one, you have a choice of which of two versions to solve. You can work on both of them if you like, but I will only give grade credit for one of them.

1 Written problems

0. Don't hand this in, but make sure you can compute by hand instances of the exclusive-or operation, for example $10011011 \oplus 01100010$.

1. You should be familiar with the operations AND and OR applied to bits. We denote these by $\wedge$ and $\vee$, and for example write $1 \wedge 0 = 0$, $1 \vee 0 = 1$. We can extend both these operations to strings of bits of the same length, so, for example,

$$11001 \wedge 01010 = 01000, 11001 \vee 01010 = 11011.$$

(In Python, these bitwise operations are denoted `&` and `|`.)

Let u, v, w be bit strings that are all the same length. Is it true *in general* that

$$u \wedge (v \oplus w) = (u \wedge v) \oplus (u \wedge w)$$

$$u \vee (v \oplus w) = (u \vee v) \oplus (u \vee w)$$

(so there are two different questions here). If the answer is no, give a counterexample. If the answer is yes, explain why this always holds.

2. Consider the monoalphabetic substitution cipher and where the set of plaintexts is $\{a, b, \ldots, z\}$, and the same cipher, but with set of plaintexts $\{aa, ab, ac, \ldots, zx, zy, zz\}$– that is, the set of all two-letter strings. (This is different from the context in which we saw the monoalphabetic substitution cipher before, where we allowed plaintexts of arbitrary length.)

Does either of these systems have perfect secrecy? Give a precise answer—that is, address the question of whether

$$\mathtt{Pr}_{k \in \mathcal{K}}[E(k, m_1) = c] = \mathtt{Pr}_{k \in \mathcal{K}}[E(k, m_2) = c]$$

for all $m_1, m_2 \in \mathcal{M}$ and all ciphertexts c. If you conclude that a system does not have perfect secrecy, you should also describe informally, in words, what kind of information the ciphertext leaks about the plaintext. (HINT: This problem has nothing to do with linguistic properties of the plaintext—so you should not use anything about distribution of letters in English text.)

3. (This is a pencil-and-paper problem, but you can use Python to do the arithmetic required in part (a).) The first random number generator I saw explicitly described was in a once-popular textbook called *Programming in Pascal,* by Peter Grogono. The generator keeps a 16-bit state and at each update outputs the entire state. The updating is done by

$$state \leftarrow (25173 \times state + 13849) \pmod{2^{16}}.$$

Suppose we use this as the keystream generator for a stream cipher.

(a) Eve intercepts four bytes of ciphertext:

```
f3 29 b0 36
```

She has strong reason to believe that the two bytes of plaintext are

```
00 ff
```

What are the next two bytes of plaintext?

(b) Alice uses this stream cipher to encrypt a draft of her new novel and send it to Bob. Explain how Eve can extract *some* information from the ciphertext even with no known plaintext. (HINT: What happens if you update the random number generator many times?)

2 Computer Problems

4. We can use the random number generator in Python as a stream cipher. (But see Problem 6 below on why you should **NEVER** do this.) We use an integer, or a string, or any hashable Python object as a key, and use this key as an argument to `random.seed` to initialize the random number generator. For instance,

```
key=random.seed("Don't share this secret with anyone!")
```

We can then construct an n-byte keystream as

```
keystream=[random.randint(0,255) for j in range(n)]
```

Some ASCII text was encrypted by this method using the secret key 'Rosebud'. The resulting ciphertext, as a base 64-encoded string, appears in the examples on the website. Find the plaintext. This is a matter of typing just a few Python commands. Include the code you typed to find the solution, along with the answer.

5. Alice Shelley uses a one-time pad to send Bob Keats two encrypted lines of her poetry. Jealous Eve Byron knows that the lines sent were from this collection of four lines, but would like to know which two Alice chose to send:

```
I met a traveller from an antique land

And on the pedestal these words appear

My name is Ozymandias---king of kings.

Look on my works ye Mighty and despair
```

(I took some liberties with the punctuation of the original so that all four lines would have exactly 38 characters.) Being more of a literature type than a computer type, Alice did not realize that she was not supposed to use the same key for both messages. Eve intercepts the ciphertexts

```
ZCkTVMy6oBNZm9QZTH2zzqU0c+PPVSf2dfoJWO8k391OTCsS5QA=
```

and

```
aD5XGsK55UdYjdQmU2C73L8xdqLIEG+oe7MQUL0vyt1EVTUQ91w=
```

and figures out which two lines Alice sent. Explain how Eve did this, and show all the calculations that led to the result.

6. *Cryptanalysis of poorly-constructed stream ciphers.* There are two versions of this problem; you need to do only one. In both cases you will be given an insecure stream cipher, the first built from a system random number generator that was not specifically designed for cryptographic use, and the second from a single LFSR. You will also be given a few hundred bytes of intercepted ciphertext, and the first several bytes of the plaintext. You can use these to recover the first few bytes of the keystream. Your job is then to exploit the weakness in design to predict the rest of the keystream, and recover the entire plaintext.

(a) One of the posted files contains a Python implementation of the random number generator in the `java.util.Random` class of Java. (In case you are wondering, it would also be possible to do this with Python's built-in random number generator, but that is somewhat more involved and requires more known plaintext.) The Java version is a somewhat old-fashioned linear congruential generator. The generator maintains a 48-bit state. You can initalize this state with a passphrase, using `init_state` (this just grabs the last 6 characters of the passphrase). The state is updated through a simple formula

$$\text{state} = (m \cdot \text{state} + k) \bmod 2^{48}.$$

Thus if we know the state at the start, we can easily generate the entire keystream.

The `nextInt()` method of Java's `Random` does not return the entire state, which would make this job easier, but only the high-order 32 bits of the state. This is implemented in the program provided by the function `next_byte`, which returns a list of 4 bytes.

To generate the rest of the keystream, we need guess the remaining 16 bits of the state. There are $2^{16} = 65536$ different possibilities to try out, so this can be done

by brute force: Initialize the state using each of the candidate values, and see if the choice successfully generates the next few bytes of the keystream. If you manage this, it means you will have recovered the state of the generator, and can then obtain all subsequent bytes of the keystream.

For this problem, the plaintext begins

```
If you
```

The character after the 'u' is a space, so all in all this gives you 7 bytes of plaintext.

The ciphertext in base 64 encoding appears on the web page accompanying this assignment.

(b) In this alternative version of the problem, the state is much smaller: A 17-bit linear feedback shift register. The register holds bits

$$b_{16}b_{15}\cdots b_1 b_0.$$

At each step, the state is updated to

$$bb_{16}b_{15}\cdots b_2 b_1,$$

where

$$b = b_{14} \oplus b_0.$$

Bit b is also the output of the generator.

The functions in the file `lfsr.py` allow you to initialize the state and get the next k bytes of output from the generator. Such a keystream, starting from an unknown initial state, was used to encrypt a passage of ASCII text. The ciphertext is given on the accompanying web page. The plaintext begins

```
He
```

If you include the final space, this gives 3 bytes=24 bits of plaintext. Coupled with the intercepted ciphertext, you get 24 bits of the keystream. You are to use this information to recover the initial state of the generator and thus find the entire keystream and decrypt the ciphertext.

It is possible to solve this problem by brute force, since there are $2^{17} \approx 130000$ keys to try out. But a more efficient method, which you should use, is to solve the underlying linear equations to find the initial state of the generator. This can even be done by hand, given the very simple recurrence for the lfsr. (But having a computer does help.) Solving requires just 17 successive bits of output

The lfsr described here has maximal period $2^{17} - 1$. It is one of two shift registers that were used in the encryption mechanism in the original Content Scrambling System used to copy-protect DVDs. (It should be added that this system is not much harder to break than a single lfsr.)