Lab 2 :: Cyphers and PRNGs

Luke Anderson: luke@lukeanderson.com.au

March 21 & 24, 2017

Previous Lab

If you haven't finished all the tasks from the previous lab –
do the previous lab first!

1 PRNGs

1.1 Linear Congruential Generators (LCG)

In the lecture, you were introduced to the linear congruential generator which calculates new pseudorandom integers by calculating $x_{n+1} = (ax + b) \bmod c$.

The period of the LCG is closely linked to the value c. The period of a PRNG is the the number of outputs before the pseudorandom sequence begins to repeat. For the LCG to have a full period, b and c must be relatively prime (i.e. their greatest common divisor is 1), $a-1$ is divisible by all prime factors of c, and $a-1$ is a multiple of 4 if c is a multiple of 4.

If these requirements aren't satisfied, the outputs of an LCG can be incredibly poor, even for non-cryptographic uses. For example, any full cycle LCG with $c = 2^n$ will alternate between odd and even outputs. This can be seen by testing the values $a, b, c = 23, 21, 2^8$ and printing the pseudorandom output $n \bmod 2$. A standard choice of parameters for an LCG used in the 1970s, called *RANDU*, were so poor that it has unfortunately called many scientific results from that period of time into question.

In the example code **lab2a_lcg.py**, Bob has decided to use an LCG for a simple gambling application. Luckily, he'll only play recreationally and won't ever involve money. Read through and understand his example code.

1.1.1 When life gives you lemons...

If you were forced to use a linear congruential generator for a project and were concerned about protecting the PRNG, what might you do? How much more secure, if at all, would this make the LCG? Specifically consider:

- How to prevent an attacker who compromises the current internal state from easily reconstructing the stream of random numbers prior to the internal state's revelation
- How to make it it difficult for the attacker from determining the internal state from the output of your PRNG

2 One Time Pads

When used correctly[1], One Time Pads (OTPs) provide perfect secrecy and provide no information about the original message to a cryptanalyst (other than the maximum possible length of the message). This is as, given a truly random key k, the resulting cyphertext c is equally likely to be any from the full cyphertext space C. It is important to note that whilst the OTP provides perfect secrecy, there are other aspects to security however. The OTP provides no guarantees as to integrity.

Given a message m and a key of the same length k, we can calculate the cyphertext by computing $c = m \oplus k$. To decode it, we just apply the key to the cyphertext, computing $m = c \oplus k = m \oplus k \oplus k = m$. This is as $x \oplus x$ cancels itself out and results purely in zeroes. If this doesn't seem intuitive, it's highly suggested you play around with XOR on a piece of paper with two arrays of bits.

Example code is provided to show how you can implement a XOR cipher yourself (**lab2b_xor.py**) and how to use the XOR cipher in the PyCrypto library (**lab2c_xor.py**).

[1]For breaking a two time pad, see "*Automatic solution in depth of one time pads*"

2.1 DES and Padding Messages

Due to the way ciphers work, ciphers commonly can only operate on messages that are a multiple of the cipher's blocksize. An example of this is the DES cipher that operates on blocks of 8 bytes. When you have a message of the wrong size, the message must be padded before it can be supplied to a cipher.

There are many ways to pad a message, but one of the easiest methods is ANSI X.923. If the message is 5 bytes long and needs to be 8, we add two null bytes and the final byte with the number of padding bytes added (in this case: 3).

$m_o = [m_1\ m_2\ m_3\ m_4\ m_5]$

$m_p = [m_1\ m_2\ m_3\ m_4\ m_5\ 0\ 0\ 3]$

In the file **crypto_utils.py**, we provide an implementation of ANSI X.923 that you may use whenever you find appropriate. You can see an example use in **lab2d_des.py**.

3 Feistel Networks

In the lectures we discussed Feistel networks, specifically through the implementation of DES[2]. The Fiestel network accepts a $2n$-bit plaintext (L_0, R_0) as input, with L_0 and R_0 being n-bit respectively. Two operations are performed at each stage in the r rounds:

$L_i = R_{i-1}$

$R_i = L_{i-1} \oplus f_{K_i}(R_{i-1})$

(where i and $i-1$ denote the current and previous iteration respectively and K_i is the subkey for the round derived from the cypher key K)

The function f_{K_i} does not need to be invertible to allow for decryption of the cyphertext. Decryption is achieved by using the same r round process but in reverse order.

[2]For more details, see Chapter 7.4 of the *Handbook of Applied Cryptography*

To see how the encryption and decryption works by working through an example of three rounds. It may look complicated but it will make sense if you work through it slowly.

ENCODING

$L_0 = m_L$

$R_0 = m_R$

—

$L_1 = R_0$

$R_1 = L_0 \oplus f_{K_0}(R_0)$

—

$L_2 = R_1$

$R_2 = L_1 \oplus f_{K_1}(R_1)$

—

$L_3 = R_2 = C_L$

$R_3 = L_2 \oplus f_{K_2}(R_2) = C_R$

—

DECODING

Now let's decode...

$L_2 = R_3 \oplus f_{K_2}(L_3) = R_3 \oplus f_{K_2}(R_2) = (L_2 \oplus f_{K_2}(R_2)) \oplus f_{K_2}(R_2) = L_2$

$R_2 = L_3$

—

$L_1 = R_2 \oplus f_{K_1}(L_2) = R_2 \oplus f_{K_1}(R_1) = (L_1 \oplus f_{K_1}(R_1)) \oplus f_{K_1}(R_1) = L_1$

$R_1 = L_2$

—

$L_0 = R_1 \oplus f_{K_0}(L_1) = R_1 \oplus f_{K_0}(R_0) = (L_0 \oplus f_{K_0}(R_0)) \oplus f_{K_0}(R_0) = L_0$

$R_0 = L_1 = m_R$

Thus, we've shown that the Feistel structure is reversible and does not rely on f_K being reversible.