Chapter 6 Notes

Common functions with bits

Here is the truth table for XOR:

A	B	$A \otimes B$
0	0	0
0	1	1
1	0	1
1	1	0

One nice thing about XOR is that it is its own inverse.

Also, for a block of bits, two typical operations are left and right shifts. Each shift can either be a regular shift or a cyclical shift.

In a right shift, all the bits move right. Here is an example:

00101101 shifted to the right by 2 bits becomes 00001011
00101101 with a cyclical shift to the right of 2 bits becomes 01001011

The key difference with a cyclical shift is that when you "move" a bit off to the right it then goes to the leftmost location.

A left shift works similarly:

01101101 shifted to the left by 2 bits becomes 10110100
01101101 with a cyclical shift to the left of 2 bits becomes 10110101

Numerically, a right shift divides by 2 and a left shift multiplies by 2.

Stream Cipher

A stream cipher requires a random "stream" of bits to use as a key, which we can then XOR with the plaintext. One reason this is desirable is that XOR is efficiently implemented in hardware. In general, we will have a key which will "start of" the random bit stream and then use it in a manner similar to the autokey cipher, where a function of the key and other information will produce the future bits.

There are two types of key generators:

1) synchronous - independent of the plaintext stream. If a ciphertext character is lost in transmission, the ciphertext and keystream will be misaligned and they will have to be realigned to recover the plaintext

2) self-synchronous - keystream produces the keystream from knowledge of the previous ciphertext characters. If there is an error in transmission, the keystream will correct itself after n correct ciphertext characters.

Linear Feedback Shift Registers

A shift register constantly performs right-shifts, but replaces the most-significant bit with the new bit in the stream.

For example if at one point in time a shift register contained 01101011 then after performing a shift, if the new bit into the stream was 1, it would contain 10110101.

This in and of itself can not create a stream of bits that seem random. We need some way of calculating the bit that gets shifted in!

In a LFSR register, the method of doing this is calculating the XOR of some specified subset of bits.

Let the contents of the register be denoted $b_8b_7b_6b_5b_4b_3b_2b_1$. For this example, define the function to determine the new bit as follows: $f(b_8b_7b_6b_5b_4b_3b_2b_1) = b_1 \otimes b_4 \otimes b_6$.

Here is an example of this LFSR:

Time	Contents
0	01101101
1	10110110
2	11011011
3	01101101
4	10110110, etc.

One can run an LFSR indefinitely, given a set of initial contents and a function to use to calculate the next bit. Obviously any LFSR will start repeating after a certain period of time. Since we know with n bits, there are 2^n configurations of those n bits, it stands to reason that the period of any LFSR of n bits is no more than 2^n. It turns out that the real maximum is $2^n - 1$, since you never want a stream with all 0s. Can you figure out why?

LFSR Period Analysis

A general LFSR function can be expressed as follows:

$$b_{n+1} = c_1b_1 \otimes c_2b_2 \otimes ... \otimes c_nb_n,$$

where $c_i = 1$, if that bit is selected for the xor function, and is 0 otherwise. In our example above, c_1, c_4, and c_6 are 1 while c_2, c_3, c_5, c_7, and c_8 are 0.

We can create a polynomial from a given LFSR as follows:

$$p(x) = \sum_{i=1}^{n} c_i x^i + 1$$

Given the characteristic polynomial of an LFSR, it turns out that the LFSR has a maximal sequence of bits if that polynomial is primitive/irreducible. (This means that the polynomial can not be factored and the smallest integer k for which p(x) divides $x^k - 1$ is $2^n - 1$.)

As an example, if we use the function $b_3 = b_1 \otimes b_3$ with a LFSR of three bits, we produce a maximal cycle of bits:

111 -> 011 -> 101 -> 010 -> 001 -> 100 -> 110 -> 111

An example of a LFSR with four bits that doesn't produce the maximal cycle is one with the following function: $b_4 = b_1 \otimes b_2 \otimes b_3 \otimes b_4$.

Here are the three separate cycles this produces:

1111 -> 0111 -> 1011 -> 1101 -> 1110 -> 1111
0001 -> 1000 -> 1100 -> 0110 -> 0011 -> 0001
1010 -> 0101 -> 0010 -> 1001 -> 0100 -> 1010

Random Bit Tests

Although it's impossible to prove if a stream of bits is random, we can check to see if a stream of bits satisfies some standard qualities of a theoretically random stream of bits. We would like our key bitstreams to be random so that Eve can't determine patterns in the key bitstream.

Here is an example of a set of test from the FIPS 140-1 poker set:

For a stream of 20,000 random bits:

1) The number of 0s must be in between 9654 and 10346

2) The distribution of the four bit segments (0 - 15) should be roughly equal. For the standard, let n_i represent the number of occurrence of i when we divide the bit stream up into 5000 blocks of 4 bits. Then we calculate a value X as follows:

$$X = \frac{16}{5000} \sum_{i=0}^{15} {n_i}^2 - 5000$$

This test passes if $1.03 < X < 57.4$.

3) A runs test sees how many runs of the same bit consecutively appear. Here are the requirements for our specific test:

Length of Run	Number of occurrences
1	2267-2733
2	1079-1421
3	502-748
4	223-402
5	90-223
6+	90-223

Breaking a Stream Cipher

Insertion Attack

Although this sort of attack is highly improbable, it does illustrate a weakness in the stream cipher.

Consider the following message:

```
p1 p2 p3 p4 p5 …
k1 k2 k3 k4 k5 …
c1 c2 c3 c4 c5 …
```

Now, imagine being able to retransmit this plaintext, encrypted with the same keystream, but by changing one bit in the plaintext and inserting it into the plaintext. Let this bit be p:

```
p1 p  p2 p3 p4 p5 …
k1 k2 k3 k4 k5 k6 …
c1 d2 d3 d4 d5 d6 …
```

Now, given the ciphertext in these two situations, we can solve for the plaintext:

$k2 = p \oplus d2$
$p2 = k2 \oplus c2$
$k3 = p2 \oplus d3$
$p3 = k3 \oplus c3,$
etc.

Consider the following example:

```
Plaintext:  p1 p2 p3 p4 p5 p6
Key:        k1 k2 k3 k4 k5 k6
Ciphertext:  1  0  1  1  0  1
```

Now, insert 1 as the second bit of the plaintext:

```
Plaintext:  p1  1 p2 p3 p4 p5 p6
Key:        k1 k2 k3 k4 k5 k6 k7
Ciphertext:  1  0  1  1  1  0  0
```

So, we first find that k2 = 1, so this yields that p2 = 1.
That means that k3 = 0, which means that p3 = 1.
Now, we have k4 = 0, so p4 = 1.
Then, we have k5 = 0, so p5 = 0.
Finally, we have k6 = 0, so p6 = 1.

Recovered plaintext bits 2 through 6: 11101.
Recovered key bits 2 through 6: 10000.

Probable-Word Attack One: Matching Bit Strings

If we have a section of matching plaintext and ciphertext, we can use the following idea to solve for the feedback function:

We know that future bits of the keystream are calculated as follows:

$$k_{m+j} = \sum_{i=0}^{m-1} a_i k_{i+j} \bmod 2$$

where m is the size of the LFSR and the coefficients a_i are the "rule" being used.

If we have m of these equations (which means obtaining 2m bits of the keystream) set up, they would look like this:

$$k_{m+1} = a_o k_1 + a_1 k_2 + a_2 k_3 + \ldots + a_{m-1} k_m$$
$$k_{m+2} = a_o k_2 + a_1 k_3 + a_2 k_4 + \ldots + a_{m-1} k_{m+1}$$
$$k_{m+3} = a_o k_3 + a_1 k_4 + a_2 k_5 + \ldots + a_{m-1} k_{m+2}$$
$$\ldots$$
$$k_{2m} = a_o k_m + a_1 k_{m+1} + a_2 k_{m+2} + \ldots + a_{m-1} k_{2m-1}$$

(Note: all equations are mod 2.)

Note that we can rewrite using matrices as follows:

$$\begin{pmatrix} k_{m+1} \\ k_{m+2} \\ \ldots \\ k_{2m} \end{pmatrix} = \begin{pmatrix} k_1 & k_2 & \ldots & k_m \\ k_2 & k_3 & \ldots & k_{m+1} \\ \ldots & \ldots & \ldots & \ldots \\ k_m & k_{m+1} & \ldots & k_{2m-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \ldots \\ a_{m-1} \end{pmatrix}$$

Since we are assuming that the key bits are known (and these can be derived from the first 2m matching bits of plain and ciphertext), we can solve for the rule (namely, the coefficients a_i as follows:

$$\begin{pmatrix} a_0 \\ a_1 \\ \dots \\ a_{m-1} \end{pmatrix} = \begin{pmatrix} k_1 & k_2 & \dots & k_m \\ k_2 & k_3 & \dots & k_{m+1} \\ \dots & \dots & \dots & \dots \\ k_m & k_{m+1} & \dots & k_{2m-1} \end{pmatrix}^{-1} \begin{pmatrix} k_{m+1} \\ k_{m+2} \\ \dots \\ k_{2m} \end{pmatrix}$$

(Remember that all calculations are done mod 2.)

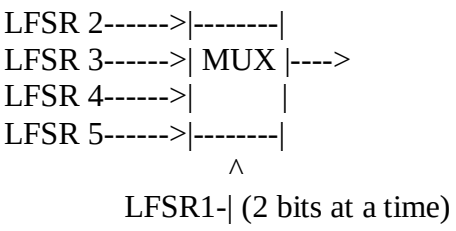
Probable-Word Attack Two: Word Match

I am skipping this section due to the fact that the attack described in the book only works if the feedback function only xor's two previous bits. (The book mentions that the attack can be generalized to more bits, but it's not clear, based on its description, how to perform this generalization.)

Other Stream Cipher Implementations

To complicate a stream cipher, one could use multiple LFSR's. One idea is the following: have several LFSR's generate their streams of pseudorandom bits. At each "cycle" select one of the several LFSR's and use the bit that it generated for that cycle as the chosen bit. In order to do this random selection, one more LFSR could be used.

The name of the device that selects one signal out of several is a multiplexer. A 4-1 multiplexer takes in 4 inputs, and selects one of the 4 as the output. Here is a picture of the scenario described above:



The bottom LFSR generates 2 bits at each cycle, while the others generate one. The two bits (00, 01, 10 or 11) determine which of the four signals above is "taken" for each clock cycle.

RC4

The algorithm varies based on a parameter n, which is typically set at 8. There are two arrays in the algorithm, an S array and a key array. Each of these arrays has $N = 2^n$, typically 256, values in it. S is initialized to store the values 0 through 255, in that order.

The key is initialized by the user to store 256 values in between 0 and 255, inclusive. Since it's cumbersome to have such a large key, typically a 4 byte key is chosen and this is repeated as necessary. (Essentially, for i > 3, key[i] = key[i-4].)

The first part of the algorithm randomizes the S array using the key as follows:

```
int j = 0;
for (i=0; i<256; i++) {
  j = (i + S[i] + K[i])%256;
  swap(S[i], S[j]);
}
```

Conceptually, what's happening here is that we loop through each element in S, and then pick a random element to swap it with in S. (j is this index, which should essentially be random...)

Once this is done, our S array is effectively permuted. At this point, we go through the following steps to produce one byte for the keystream (note: we start i and j back at 0.)

```
i++;
j = (j+S[i])%256
swap(S[i],S[j]);
t = (S[i]+S[j])%256;
k = S[t];
```

The byte produced is k, which is the value stored in S at index t. Index t is computed as the sum of seemingly random elements in S. Notice that the swap does NOT change the value of t, but the swap makes sure that the array is changing constantly.

To add more bytes to the keystream, simply repeat the steps above as many times as necessary (without resetting i or j again).

Our text simply mentions that doing a brute force search on RC4 is not feasible. There are some keys (2^{-2n} fraction of them) that are impossible, but this reduces the search space in a very minor way. An attack on a 40-bit RC4 key has been performed in 159 days. This makes it a feasible threat, but of course, one could always use a larger key very easily.

A5

The keystream created by A5 consists of three LFSRs, of sizes 19 bits, 22 bits and 23 bits. The output stream is the XOR of the outputs of these three registers.

It is an irregularly clocked system, which simply means that not each of the registers shift at each time cycle. Rather, each one shifts some times and not other times, and when each one shifts is not easily correlated to the other two.

The picture on page 122 of the text captures the operation of the system rather well.

The three registers are initialized with the key (64 bits), and then the system is run. The output is always XOR(A[18], B[21], C[22]). To advance to the next clock cycle, we calculate MAJ(A[9],B[11],C[11]), then this value is XORed with A[9], B[11] and C[11], individually to see whether or not A, B or C shift. (Any subset of them could theoretically shift on a single time step.)

A couple weaknesses of A5 are that by knowing the contents of the registers A and B, a known-plaintext attack would compromise the values in C. Also, multiple initial positions lead to the same keystream, so there are fewer than 2^{64} possible key streams.

A5 is used to encrypt voice communication on GSM systems, in 228 bit blocks.

Cellular Automata

A cellular automata is an array, which changes at each time step based on some sort of rule. If applied to an infinite array, these changes could be indefinite. On a finite array, they create a cyclic pattern, just like an LFSR.

Here's a basic example with a 7-bit Cellular Automata:

Let the following be the initial contents:

index	0	1	2	3	4	5	6
value	0	0	1	0	1	0	0

Now, we must define a "rule" which describes how the automata changes. Our rule will involve the bit itself and its immediate neighborhood. Here is an example of a rule:

neighborhood	000	001	010	011	100	101	110	111
new value	0	0	0	1	0	1	1	1

Thus, when deciding what to change a bit A[i] to in time step t+1, look at the three bits A[i-1], A[i] and A[i+1] in time step t. Look up this neighborhood on the chart above, and then place the new value accordingly. Here is what would happen to the automata above for the next two time steps

index	0	1	2	3	4	5	6
t=0	0	0	1	0	1	0	0
t=1	0	0	0	1	0	0	0
t=2	0	0	0	0	0	0	0

(from this point, this will just be all 0s)

Now, let's try another rule:

neighborhood	000	001	010	011	100	101	110	111
new value	1	0	1	1	0	1	0	1

Here are a few time steps with the same initial starting conditions:

index	0	1	2	3	4	5	6
t=0	0	0	1	0	1	0	0
t=1	1	0	1	1	1	0	1
t=2	1	1	1	1	0	1	1

To generate a stream of "random bits" we can simply select one bit in the cellular automata (for example, bit 5) at each time step.

The behavior of two dimensional systems can be significantly more complex than of one dimensional systems.