Chapter 6 : Modern symmetric encryption.

First, from here on we will work in binary.
So we'll have to think in sums of powers of 2 instead of powers of 10 from here on out.

E.g. $37 = 2^5 + 2^2 + 2^0$
$= 100101$
or $7 = 2^2 + 2^1 + 2^0$
$= 111$

Of course, we can still encrypt messages involving letters/punctuation etc, we just have to decide on a correspondence between letters/punctuation and binary numbers. Thankfully this is already done for us.

E.g. ASCII

American Standard Code for Information Interchange.
Each letter corresponds to a 5-digit (or 5-bit) binary string.

E.g: HI becomes $\underbrace{00111}_{H}\ \underbrace{01000}_{I}$.

The benefit of binary lies in the simplicity of operations on binary numbers and strings.

There's good old addition:

$$\begin{array}{r} \overset{1}{1}\,\overset{1}{1}\,0\,1 \\ +\quad 1\,0\,1 \\ \hline 1\,0\,0\,1\,0 \end{array}$$

or in more familiar notation:

$$\begin{array}{r} 13 \\ +\ 5 \\ \hline 18 \end{array}$$

$= 2^4 + 2^1 = 18$ (it checks out).

There's also "bitwise addition" where we just add the entries mod 2, with no carrying (ie use: $0+0=0$, $1+0=0+1=1$, $1+1=0$)

Also called XOR (eXclusive Or). It's like this; written $\oplus$:

$$\begin{array}{r} 001011101 \\ \oplus\ 101111100 \\ \hline 100100001 \end{array}$$

One of our first goals will be to use binary to generate from a relatively small key a long random-seeming string of 0's and 1's, useful in stream ciphers (for example).

§6.2 Linear feedback shift registers.

Our notation will be the same as the book, meaning we have to think right-to-left.

An LFSR is a way of creating a new string of 0's and 1's from a set of "transition rules".

Example: We are going to transform the 4-bit string (b_4', b_3', b_2', b_1') into (b_4, b_3, b_2, b_1). We write this as

$$(b_4, b_3, b_2, b_1) \longleftarrow (b_4', b_3', b_2', b_1').$$

To describe the change, we write "transition rules":

$$b_3 \longmapsfrom b_4' \qquad b_2 \longmapsfrom b_3'$$
$$b_1 \longmapsfrom b_2' \qquad b_4 \longmapsfrom b_1' + b_2'.$$

To make this notation make sense, we can read the arrow "$\longmapsfrom$" as "comes from". So, with the rules above,

1011 corresponds to $b_4' = 1$, $b_3' = 0$, $b_2' = 1$, $b_1' = 1$

and it is transformed into .

0101

$b_3 \longmapsfrom b_2'$, etc . . .

$b_4 \longmapsfrom b_1' + b_2'$

In general, a LFSR is a single formula, such as

$$b_5 \longmapsfrom b_1' + b_2' + b_4'$$

where the subscript on the left indicates the size of the string, and the formula indicates the only "non-trivial" transition rule. I.e. all other transition rules are of the form $b_i \longmapsfrom b_{i+1}'$.

An initial value used to compute subsequent strings is called the <u>seed</u> for the register.

<u>Example continued</u> : Using the LFSR $b_4 \longmapsfrom b_1' + b_2'$ above, and a seed of 0010, we can compute a table of subsequent strings:

Step	string
0	0 0 1 0 ← seed.
1	1 0 0 1
2	1 1 0 0
3	0 1 1 0
4	1 0 1 1
5	0 1 0 1
6	1 0 1 0
	⋮

note this column may seem totally random to someone who does not know the rule $b_4 \longleftarrow b_1' + b_2'$.

The column on the right will eventually repeat, since there are at most $1111 = 2^3 + 2^2 + 2^1 + 2^0 + 1 = 16$ arrangements of four zeroes and 1's. We can use these bitstreams for stream ciphers.

<u>Example</u>: Consider the LFSR $b_4 \longleftarrow b_1' + b_2' + b_4'$, seeded with 0110. We start computing states, and get:

step	string
0	0 1 1 0
1	1 0 1 1
2	1 1 0 1
3	0 1 1 0
4	1 0 1 1
5	⋮

repeats soon!

Now to encrypt a message like "HELP" we write it in binary using ASCII, get:

	H	E	L	P
plaintext	0 0 1 1 1	0 0 1 0 0	0 1 0 1 1	0 1 1 1 1
keystream $\oplus$ XOR	0 1 1 0 1	1 0 1 1 0	1 1 0 1 1	0 1 1 0 1
ciphertext	0 1 0 1 0	1 0 0 1 0	1 0 0 0 0	0 0 0 1 0
change back:	K	S	Q	C .

To decrypt: Pass the rule $b_4 \longleftarrow b_1' + b_2' + b_4'$ to the intended recipient. They generate the same keystream:

0 1 1 0 1 1 0 1 1 ... etc.

Then XOR the ciphertext with the keystream, taking advantage of the fact that subtraction mod 2 is the same as addition mod 2! So:

ciphertext	0 1 0 1 0 1 0 0 ....
keystream $\oplus$	0 1 1 0 1 1 0 1 ....
plaintext	0 0 1 1 1 0 0 1 ... etc.

Remark: If we chose a LFSR with many bits, then it might not repeat for a long time, unlike the previous example. In general, a LFSR

$$b_k \longleftarrow \text{some equation}$$

doesn't necessarily repeat before the 2^k-th step (though some might repeat much sooner). So we could potentially communicate only a simple equation and a length k seed, and generate a string $\sim 2^k$ bits long that is "effectively random" to an observer.

In fact, we could make the formula public! The seed alone will generate a keystream for the intended recipient, and an eavesdropper would still be lost as to how to generate the keystream.

§6.3 : A vulnerability: Known plaintext attacks.

Recall we want a system where an eavesdropper that somehow discovers a large chunk of our plaintext will still be thwarted.

So suppose that someone knows the LFSR we're using (ie the formula). We send the ciphertext

WIMSUJ

using $b_4 \leftarrow b_4' + b_1'$. Our seed is secret.
Unfortunately, an eavesdropper finds out that the first two letters are "MR".

ciphertext : 1 0 1 1 0 0 1 0 0 0 ....

they know : 0 1 1 0 0 1 0 0 0 1 .... ← plaintext.

Since addition mod 2 is the same as subtraction, they need only XOR these to get the start of the keystream:

keystream: 1 1 0 1 0 1 1 0 0 1

So if we imagine filling in our LFSR table:

step	String			
0	*	*	*	1
1	*	*	*	1
2	*	*	*	0
3		?		1
4				0
5				1
6				1
				etc.

to crack our message, they only need to fill in this rightmost bit for each state.

~~the~~ They can reverse engineer the whole seed knowing this portion of the table, unfortunately. Recall the "unstated rules" are $b_i \longleftrightarrow b'_{i+1}$ for all i. So:

Step					
	1	0	1	1	there's the seed.
		1	0	1	this entry comes from the one to the top left
			1	0	
				1	

So it takes nothing to now generate the keystream by working forward! In fact this means that <u>in general</u>: The first k ~~elements~~ bits in the keystream generated by a k-bit LFSR <u>are</u> the seed. So a very small portion of plaintext is needed to crack the code.

§6.4. LFSRsum: A fix to LFSR keystreams, and known plaintext attacks.

LFSRsum is the name we will give to a "baby example" of our eventual fix.

Idea: Use a 3-bit LFSR and a 5-bit LFSR in tandem.

The 3-bit LFSR will be: $b_3 \longleftarrow b_2' + b_1'$ and the 5-bit LFSR will be $c_5 \longleftarrow c_4' + c_2' + c_1'$.

$\boxed{\text{New notation: From here on we'll write "=" instead of } \longleftarrow \text{, and understand it to mean the same thing.}}$

- We adopt the convention that the seed for the registers is never 0, so a seed for each register will always end in "1" just to be safe.

Now to generate the LFSRsum keystream, choose two seeds 0 1 1 and 1 0 1 0 1, then:

LFSR-3

step	string		
0	0	1	1
1	0	0	1
2	1	0	0
3	0	1	0
4	1	0	1
5	1	1	0
6	1	1	1
⋮	⋮		

LFSR-5

step					
0	1	0	1	0	1
1	1	1	0	1	0
2	0	1	1	0	1
3	0	0	1	1	0
4	etc				1
5					1
6					0
⋮					

sum these.

We create the new keystream:

$$\begin{array}{r} 1\ 1\ 0\ 0\ 1\ 0\ 1\ \ldots \\ \oplus\ 1\ 0\ 1\ 0\ 1\ 1\ 0\ \ldots \\ \hline 0\ 1\ 1\ 0\ 0\ 1\ 1\ \ldots \end{array}$$

Then encrypt our message by XORing with this.

Our key for this whole scheme, given our convention of having the keys for each register end in "1", is only 5 bits:

$$\underbrace{0\ 1}_{\text{first two bits of LFSR-3 seed}} \qquad \underbrace{1010}_{\text{first four bits of LFSR-5 seed.}} \qquad (6 \text{ bits total})$$

Can we use a known plaintext attack here?

Assume we have the ciphertext. We figure out the first bit of the plaintext, and XOR it with the ciphertext to get the start of the keystream:

$$0\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ \underbrace{\ldots\ldots}_{\text{unknown.}}$$

Let's say the 6-bit key used to create this keystream from LSFRsum is

$$k_1 k_2 k_3 k_4 k_5 k_6$$

where each k is 0 or 1. How to solve fore the k's?

Now let's use the seeds $k_1 k_2 1$ and $k_3 k_4 k_5 k_6 1$ in the LFSR-3 and LFSR-5 registers.

LFSR-3

step	string		
0	k_1	k_2	1
1	$1+k_2$	k_1	k_2
2	k_1+k_2	$1+k_2$	k_1
3	$1+k_1+k_2$	k_1+k_2	$1+k_2$
4		$1+k_1+k_2$	k_1+k_2
5			$1+k_1+k_2$
⋮			$1+k_1$
⋮		etc	1
			k_2
			k_1

keystream of LFSR-3

LFSR-5

step	string				
0	k_3	k_4	k_5	k_6	1
1	$1+k_6+k_4$	k_3	k_4	k_5	k_6
2	$k_6+k_5+k_3$	$1+k_6+k_4$	k_3	k_4	k_5
3		$k_6+k_5+k_3$	$1+k_6+k_4$	k_3	k_4
4				$1+k_6+k_4$	k_3
⋮					$1+k_6+k_4$
⋮					$k_6+k_5+k_3$
		etc.			

keystream

What happens here? Well, in proceeding to the nexts steps we do $(k_1+k_2)+(1+k_2) = 1+k_1+\underset{0 \bmod 2}{\underset{\shortparallel}{2k_2}} = 1+k_1$

and similarly $1+(k_1+k_2)+(k_1+k_2) = 1 \bmod 2$.

We know that summing these keystreams must give the start of our key, which we know to be:

$$0\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1 \ldots \text{etc}$$

because of the known plaintext.

So we get equations:

LFSR-3	LFSR-5	keystream
1	$+ \quad 1$	0 ✓
k_2	$+ \quad k_6$	1
k_1	$+ \quad k_5$	1
$1+k_2$	$+ \quad k_4$	1
k_1+k_2	$+ \quad k_3$	1
$1+k_1+k_2$	$+ \quad 1+k_6+k_4$	0
$1+k_1$	$+ \quad k_6+k_5+k_3$	0
	$\vdots$	

At this point, we have 6 linear equations in 6 unknowns. This means that if there is a solution, we've got enough information to find it (in general you need n linear eqns in n variables to uniquely determine a solution - if there is one). In linear algebra terms, we've arrived at:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \\ k_6 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

and we can solve this system by (for example) row reduction. Thus we can determine the key $k_1k_2k_3k_4k_5k_6$ and then generate the keystream to decode the message, so again this succumbs to a known plaintext attack.