Stream Cipher

A stream cipher encrypt bits individually, that is to say, bits are processed one by one.

The XOR operation

message bit	key bit	xor result(cipher bit)
0	0	0(unchanged)
1	0	1(unchanged)
0	1	1(flipped)
1	1	0(flipped)

This table shows us that as long as we generate the key **randomly**, the chance of message bit to be flipped is always 50%(xor 0 leave the bit unchanged, xor 1 flip it), which is perfectly meet **Shannon theorem**, the cipher bits do not give out any information about everything

---

Generate the key

This is the heart of stream cipher, this is stuff about randomness

Random Number Generator(RNG)

1.True Random Number Generator(TRNG)

- Flipping coin is a perfect example of TRNG
- Random physical operations are often good resources
  - enviroment noise
  - Interval between key stroking
  - mouse movement
  - .....
- True random number cannot be recreated

2.Pseudo Random Number Generator(PRNG)

- Pseudo random numbers are computed, that is to say that they are deterministic
- they are usually generated by a function

$$s_0 = \text{seed}$$

$$s_{i+1} = f(s_i)$$

- example:

  in ANSI C(-std=c90), the theory works behind `rand()` function is(Only for type 0 random number)

$$s_{i+1} = 1103515245 \times s_i + 12345 \mod 2^{31}$$

```
//part of the source code of rand()
 if (buf->rand_type == TYPE_0)
    {
      int32_t val = ((state[0] * 1103515245U) +
12345U) & 0x7fffffff;
      state[0] = val;
      *result = val;
    }
else
    {
      int32_t *fptr = buf->fptr;
      int32_t *rptr = buf->rptr;
      int32_t *end_ptr = buf->end_ptr;
      uint32_t val;
      val = *fptr += (uint32_t) *rptr;
      /* Chucking least random bit.  */
      *result = val >> 1;
      ++fptr;
      if (fptr >= end_ptr)
        {
          fptr = state;
          ++rptr;
        }
      else
        {
          ++rptr;
          if (rptr >= end_ptr)
            rptr = state;
        }
```

```
      buf->fptr = fptr;
      buf->rptr = rptr;
    }

printf("%d" , rand())
//1804289383

//This is calculated by
signed int fptr = 1040273694
signed int rptr = -1726662223
unsigned int result = (unsigned int)fptr + (unsigned
int)(rptr)
//result now is 2568305073
return result >> 1
//that 2568305073 >> 1 == 1804289383
```

If you want to know more about **rand.c**, click the link.

(Note that 1804289383 is the result of Linux gun gcc, for MacOS this is not the case)

3.Cryptographical Secure Pseduo Random Numbers Generator(CPRNG)

CPRNG has an important property : the numbers are unpredictable

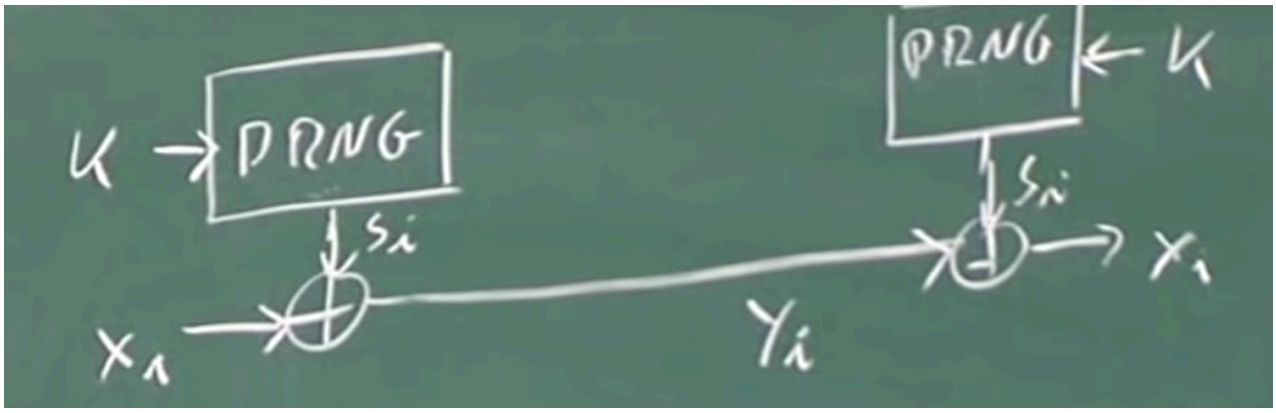
Unpredictable means given n output bits $S_0, . . , S_i, S_{i+1} . . . S_n$, it is computationally infeasible to construct S_{n+1}

One Time Pad(OTP)

> A cipher is "unconditionally" secure if it can not be broke even with infinite computation resources

> ### Definition
>
> The OTP is a stream cipher where
>
> - the key stream bits s_i stem from a PRNG
> - Each key bit is used only once
>
> However, they are difficult to use, the key have the same length of the message, and each time after using the key, you need to destroy it.

Linear Congruential Generator



This is how modern cryptography is applied in practice

$$S_0 = \text{seed}$$

$$s_{i+1} \equiv A \times s_i + B \mod m,\ A, B, m \in Z_m$$

$$\text{The key is } (A, B)$$

However, it the cipher acutally work like this, it will be easy to break it.

say that the hacker got 3 pairs of $(\text{message}\ ,\ \text{cipher})$

Then he can calculate the key by the following

1. $s_i = m_i \oplus s_i \oplus m_i = c_i \oplus m_i$
2. after we get 3 s_i, let's name it s_1, s_2, s_3

$$\begin{cases} s_2 \equiv A \times s_1 + B \mod m \\ s_3 \equiv A \times s_2 + B \mod m \end{cases}$$

just solve the equations, the K(A,B) is released.

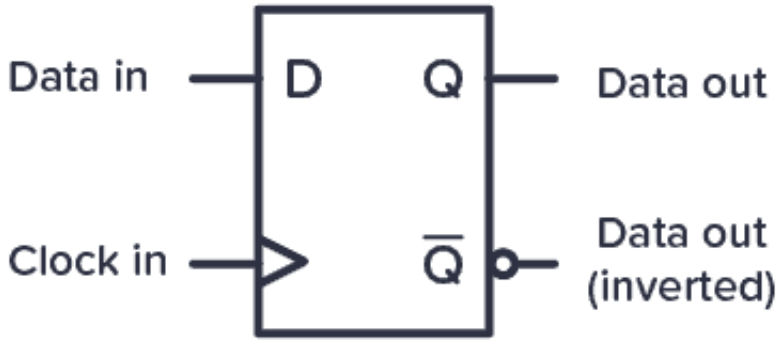
$$\begin{cases} A \equiv (s_3 - s_2) \cdot (s_2 - s_1)^{-1} \mod m \\ B \equiv s_2 - (s_3 - s_2) \cdot (s_2 - s_1)^{-1} \cdot s_1 \mod m \end{cases}$$

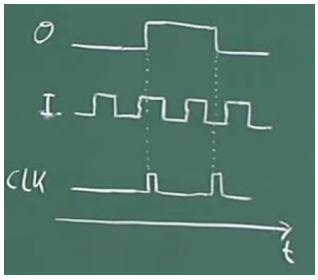
3. Once you get (A,B) , you can calculate $s_4, s_5, \ldots, s_i, s_{i+1}$

So, this is not secure, a hacker can easily get 3 pairs of (message , cipher) in something like a package header, protocol header..... What we really do in the real world is LFSR(Linear Feedback Shift Registers)

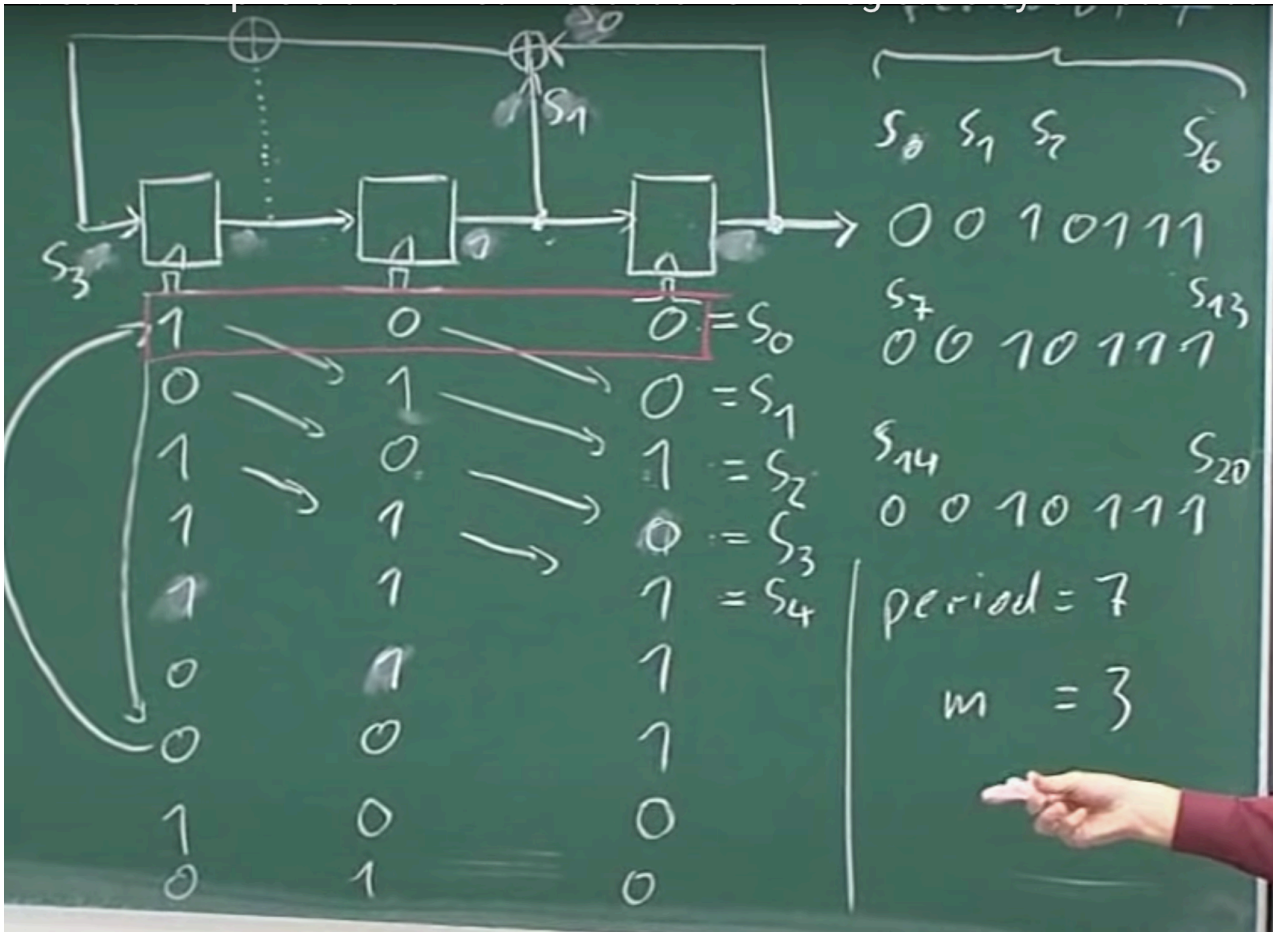
LFSR(Linear Feedback Shift Registers)

LFSR is consist of flip-flops, which can store the data go through it, controlled by the clock signal



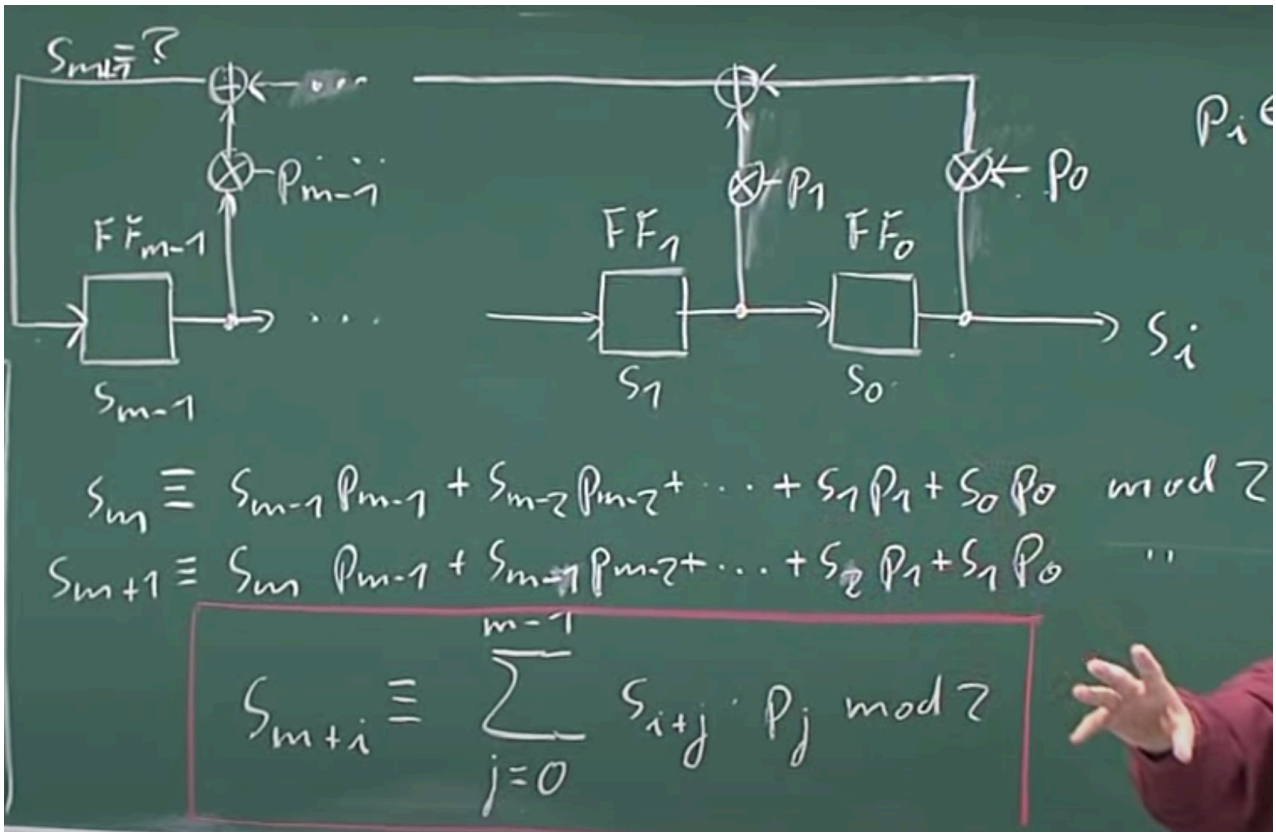


How the LFSR work



The key part is how the feedback is generated.

The designer should decide this, use p_i to represent whether the output of a specific flip-flop is used or not.



> Theorem:
>
> Only LFSRs with primitive polynomial representation yield max length period

The numbers generated by LSRF are random from a **statistic** perspective, but we do not have unpredictable at all !

Attack

The hacker need to know

- the cipher
- the number of flip-flops used in the LSRF, that is, the degree m

- the first 2m bits of the message (from the header information)

Let the attack begin

1. compute the $s_i, (i = 0, 1, \ldots, 2m-1)$,
   $s_i = m_i \oplus s_i \oplus m_i = c_i \oplus m_i$
2. write out the equations

$$s_m = s_{m-1}p_{m-1} + s_{m-2}p_{m-2} + \ldots s_0 p_0 \mod 2$$

$$s_{m+1} = s_m p_{m-1} + s_{m-1}p_{m-2} + \ldots s_1 p_0 \mod 2$$

$$\ldots\ldots$$

$$s_{2m-1} = s_{2m-2}p_{m-1} + s_{2m-3}p_{m-2} + \ldots s_{m-1}p_0 \mod 2$$

3. sovle it(by matrix invertion or Gauss elimination), then you can get all p_i, that is to say that you can now build a same LFSR to generate the rest of the keys

---