

# Lesson 2021-03-09

March 14, 2021

## Stream ciphers

Let's consider a typical cryptography application: GSM. We have communication between the MT (mobile terminal) and the BS (base station) through analog signals, which contain our voice. While going from the MT to the BS our voice is encrypted with stream ciphers.

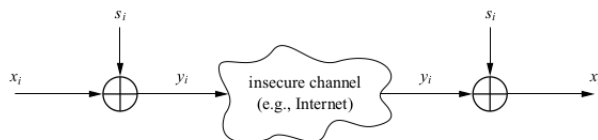
A stream cipher concerns streams of bits processed one by one, then encrypted *individually*. How does encryption/decryption work?

$$y_i = Enc(x_i) \equiv x_i + s_i \text{ mod } 2$$

$x_i = Dec(y_i) \equiv y_i + s_i \text{ mod } 2$  Where  $y_i, x_i, s_i \in \mathbb{Z}_2 = \{0, 1\}$ , so they are bits. Why do both encryption and decryption have a plus even if they are opposite operations? This is a property of modular arithmetic: mod2 addition and subtraction are the same operation because they correspond to the XOR operator  $\oplus$

$x_i$	$s_i$	$y_i = x_i \oplus s_i$
0	0	0
0	1	1
1	0	1
1	1	0

Stream ciphers use the XOR operator because its truth table gives 0 or 1 with the same probability. Observation: if we XOR a bit with 0 it remains unchanged, meanwhile with 1 the bit flips.



That's it? Have we finished with cryptography? Well, we forgot the hotpot, i.e how do we generate key bits? Randomness!

# 1 Randomness

Randomness is found everywhere in cryptography: in the generation of secret keys, in encryption schemes, and even in the attacks on cryptosystems. Without randomness, cryptography would be impossible because all operations would become predictable, and therefore insecure. Randomness is found everywhere in cryptography: in the generation of secret keys, in encryption schemes, and even in the attacks on cryptosystems. Without randomness, cryptography would be impossible because all operations would become *predictable*, and therefore insecure. Bear in mind: predictability is crucial in cryptography!

Any randomized process is characterized by a *probability distribution*, which gives all there is to know about the randomness of the process. A probability distribution, or simply distribution, lists the outcomes of a randomized process where each outcome is assigned a probability.

A probability measures the likelihood of an event occurring. It's expressed as a real number between 0 and 1 where a probability 0 means impossible and a probability of 1 means certain. For example, when tossing a two-sided coin, each side has a probability of landing face up of  $1/2$ , and we usually assume that landing on the edge of the coin has probability zero.

We have 2 types of RNG (Random Number Generator): TRNG and PRNG

## 1.1 TRNG (True Random Number Generator)

The first example of a TRNG that comes in mind is the trivial process of flipping a coin  $n$  times, in order to obtain a sequence of  $n$  values (heads or tails). The greater is  $n$ , the harder is to replicate the sequence. In computing, a hardware random number generator (HRNG) or true random number generator (TRNG) is a device that generates random numbers from a physical process, rather than by means of an algorithm. Such devices are often based on microscopic phenomena that generate low-level, statistically random "noise" signals, such as thermal noise, the photoelectric effect, involving a beam splitter, and other quantum phenomena. These stochastic processes are, in theory, completely *unpredictable*, and the theory's assertions of unpredictability are subject to experimental test. This is in contrast to the paradigm of pseudo-random number generation commonly implemented in computer programs.

A hardware random number generator typically consists of a transducer to convert some aspect of the physical phenomena to an electrical signal, an amplifier and other electronic circuitry to increase the amplitude of the random fluctuations to a measurable level, and some type of analog-to-digital converter to convert the output into a digital number, often a simple binary digit 0 or 1. By repeatedly sampling the randomly varying signal, a series of random numbers is obtained.

The main application for electronic hardware random number generators is in cryptography, where they are used to generate random cryptographic keys to transmit data securely. They are widely used in Internet encryption protocols such as Transport Layer Security (TLS).

## 1.2 PRNG (Pseudo Random Number Generator)

In this case the random sequence is computed, i.e it's deterministic. This makes the sequence not so random after all. How are these sequences computed? Let's give a more specific definition.

A PRNG is a function  $G : \mathbb{Z}_2^n \rightarrow \mathbb{Z}_2^{l(n)}$ ,  $l(n) > n$  (expansion factor), such that no adversary  $A$  succeed with probability  $> 1/2$  the so called PRNG indistinguishability experiment  $PRG_{A,G}(n)$ :

- a) A uniform bit  $b \in \{0, 1\}$  is chosen. If  $b = 0$  then choose a uniform  $r \in \{0, 1\}^{l(n)}$ . If  $b = 1$  then choose a uniform  $s \in \{0, 1\}^n$  and set  $r := G(s)$ .
- b) The adversary  $A$  is given  $r$ , and outputs a bit  $b'$ .
- c) The output of the experiment is defined to be 1 if  $b'=b$ , and 0 otherwise.

The function  $G$  expands  $n$  bits  $s \in \mathbb{Z}_2^n$  into a long sequence  $G(s)$  of  $l(n)$ -bits.

In general we can construct a PRNG from two algorithms (Init, GetBits):

- Init periodically takes as input a seed  $s$  and an optional IV (initialization vector) from a TRNG (i.e an entropy source). It uses them to update the *entropy pool* (basically a memory buffer) and outputs an initial state  $st_0$ .
- GetBits takes an input state  $st_i$ , outputs a bit  $y$  and updates the state to  $st_{i+1}$ .

Let's have a look to a pseudo-code implementation:

```
st0 = Init(s,IV)
for i = 1 to l(n):
  (yi, sti) = GetBits(sti-1)
return y1, ..., yl(n)
```

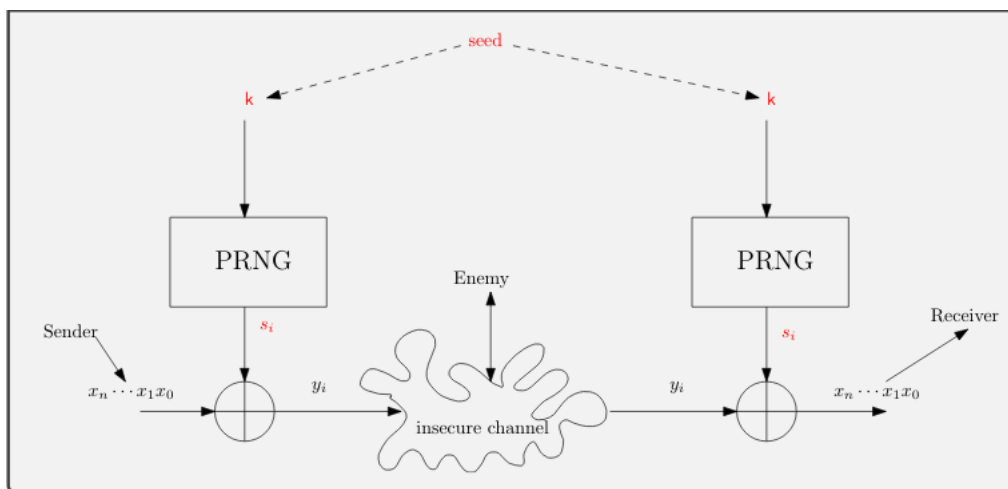


Figure 1: Representation of a stream cipher (we'll discuss later about it) using a PRNG. The plaintext is represented by  $x_n \dots x_1 x_0$  bits, while the PRNG generates  $s_n \dots s_1 s_0$  random bits and  $y_n \dots y_1 y_0$  are the ciphertext bits.

To construct a scheme secure for encrypting multiple messages, we must design a scheme in which encryption is *randomized* so that when the same message is encrypted multiple times, different ciphertexts can be produced. This may seem impossible since decryption must always be able to recover the message. However, we will soon see how to achieve it.

### 1.3 CPRGNG (Cryptographic Pseudo Random Number Generator)

This is the cryptographically secure version of PRNG (so far we've just analyzed how random numbers are generated, we don't have spoken about cryptography yet). There are both cryptographic and non-cryptographic PRNGs. Non-crypto PRNGs are designed to produce uniform distributions for applications such as scientific simulations or video games. However, you should never use non-crypto PRNGs in crypto applications, because they're insecure. They're only concerned with the quality of the bits' probability distribution and not with their predictability. Crypto PRNGs, on the other hand, are *unpredictable* (do you remember this key concept?), because they're also concerned with the strength of the underlying operations used to deliver well-distributed bits.

PRNGs can work in two different modes: synchronized and unsynchronized

## 1.4 PRNG: Synchronized mode

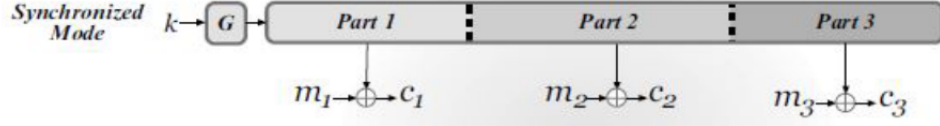


Figure 2: The key is represented by  $k$ ,  $m_i$  are the plaintexts while  $c_i$  are the ciphertexts. Note that in this mode the stream cipher doesn't need the IV

Using (Init, GetBits) the parties construct  $G(k, 1^l)$  running  $l$ -times using a shared key  $k$ . Concretely both begin by computing  $st_0 = \text{Init}(k)$ .

To encrypt the first message  $m_1$  of length  $l_1$ , the sender runs GetBits  $l_1$ -times beginning at  $st_0$  getting bits  $y_1 \dots y_{l_1} = \text{pad}_1$  along with an updated state  $st_{l_1}$ .

It sends  $c_1 = \text{Enc}_l(m_1) = \text{pad}_1 \oplus m_1$ .

Once the other party receives  $c_1$  it runs GetBits  $l_1$ -times beginning at  $st_0$  getting bits  $y_1 \dots y_{l_1} = \text{pad}_1$  along with an updated state  $st_{l_1}$ . It uses  $\text{pad}_1$  to recover  $m_1 = c_1 \oplus \text{pad}_1$ .

Later the sender, to encrypt a second message  $m_2$ , will run GetBits at the state  $st_{l_1}$  to obtain a second  $\text{pad}_2 = y_{l_1+1} \dots y_{l_1+l_2}$  and an updated state  $st_{l_1+l_2}$  and so on.

In this case we use a different part of the output stream to encrypt each new message, then sender and receiver need to know which position is used to encrypt each message.

## 1.5 PRNG: Unsynchronized mode

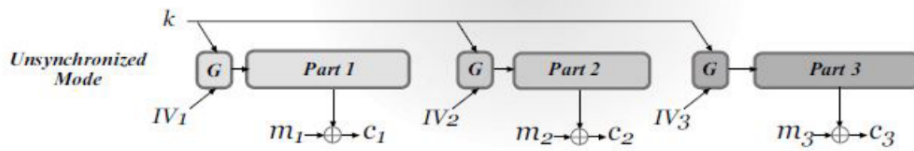


Figure 3:

In this case ciphertexts are pairs:

$$\text{Enc}_k(m_j) = \langle IV_j, G(k, IV_j, 1^{|m_j|}) \oplus m_j \rangle \quad (1)$$

where  $G(k, IV_j, 1^{|m|})$  is the pad obtained by running  $\text{Init}(k, IV_j)$ . IV must be randomly chosen, and freshly chosen for each message.

Decryption is performed in the natural way. Note: typical mistakes are related to the optional IV e.g a fixed constant in soft or hardware. If IV is random then  $Enc_k(m) = \langle IV, G(k, IV, 1^{|m|}) \oplus m \rangle$  is CPA-secure (chosen plaintext attacks), but not CCA-secure (chosen ciphertext attacks).

- CCA: The cryptanalyst can choose different ciphertexts to be decrypted and has access to the decrypted plaintext.
- CPA: The cryptanalyst not only has access to the ciphertext and associated plaintext for several messages, but he also chooses the plaintext that gets encrypted.

**Exercise:**

Let  $G(k, IV, 1^{|m|})$  as before and consider the following symmetric encryption scheme:

$k$  is random generated by Gen and shared by the parties.

Enc: on input a key  $k$  and a message  $m$  choose a random IV and output:

$$Enc_k(m) = \langle IV, G(k, IV, 1^{|m|}) \oplus m \rangle \quad (2)$$

Dec: on input a key  $k$  and a ciphertext  $\langle IV, c \rangle$  output the plaintext message:

$$Dec_k(\langle IV, c \rangle) = G(k, IV, 1^{|c|}) \oplus c = m \quad (3)$$

Show that the above is not CCA-secure. Hint: an adversary can guess the bit  $b$  using  $m_0$ = all zeros and  $m_1$ = all ones and a CCA-oracle query.

**Solution:**

Since the IV is sent in clear, the eavesdropper can obtain it. In a CCA the attacker chooses a ciphertext and has access to an oracle which can decrypt it but the attacker can't recover the key (which is actually the goal). Let's say that the attacker chooses any 2 messages  $m_0$  (e.g all zeros) and  $m_1$  (e.g all ones) of equal length. Let the challenge ciphertext be  $\langle IV, c \rangle$  where  $c := G(k, IV) \oplus m_b$  with  $b \in \{0, 1\}$ .

The attacker modifies the ciphertext to  $\langle IV, \bar{c} \rangle = \langle IV, G(k, IV) \oplus \bar{m}_b \rangle$  which is a legitimate ciphertext for  $\bar{m}_b$ . Requesting the oracle to decrypt  $\langle IV, \bar{c} \rangle$ , the attacker will get  $\bar{m}_b$  and hence the value of  $b$ .

## 2 The Lehmer generator and LCG (Linear Congruence Generator)

A linear congruential generator is a method of generating a sequence of pseudorandom numbers. Furthermore the idea is to use a key stream  $s_i$  from a PRNG. An LCG generates pseudorandom numbers by starting with a value called the seed, and repeatedly applying a given recurrence relation to it to create a sequence of such numbers.  $s_0$ = seed

$$s_{i+1} \equiv a \cdot s_i + b \pmod{m}, i = 0, 1, \dots \quad (4)$$

Where  $a$  and  $b$  are constants and  $a, b, s_i$  are  $\log_2 m$  bits long. A well-known example is *rand()* of ANSI C, where  $s_0 = 12345$  and  $s_{i+1} \equiv 1103515245 \cdot s_i + 12345 \pmod{2^{31}}$ ,  $i = 0, 1, \dots$

**Exercise:**

Setting  $b=0$ ,  $s_0 = 47594118$ ,  $a=23$  and  $m = 10^8 + 1$ , show that each  $s_n$  is divisible by 17. It is possible to show that the number 23 is the best choice in the sense that no other number produces a longer period, and no smaller number produces a period more than half as long.

**Solution:**

Since  $b = 0$ , then  $s_{i+1}$  can be computed as

$$s_{i+1} = a \cdot s_i \pmod{10^8 + 1} = a^{i+1} \cdot s_0 \pmod{10^8 + 1} \quad (5)$$

Here's a code example of a Lehmer LCG (also called Park Miller RNG):

```
s = 1
a = 7**5
tp=2**31
m=tp-1
i=1
while i < 10002:
    print(i,s) s = (a*s)%m
    i+=1
```

### 3 LFSR (Linear Feedback Shift Register)

This is another way to generate pseudo random streams of bits. The LFSR is a shift register whose state transition is linear.

Basically a shift register is a concatenation of atomic elements called *flip flops*, which are hardware components that store the single bit.

- The state  $s$  is a row vector of  $m$  bits, i.e  $s \in \{0, 1\}$ .
- The sequence of bits in the rightmost position of the state is the **output stream** or key stream.
- The bits in the LFSR state that influence its behaviour are called **taps**.

A **maximum-length** LFSR produces an  $m$ -sequence (i.e it cycles through all possible  $2^m - 1$  states within the shift register except the state where all bits are zero), unless it contains all zeros, in which case it will never change.

Two given LFSR are mirror LFSR if their characteristic polynomial are reciprocal between them. The output stream of a LFSR is reversible and given by the output stream of its reciprocal LFSR. So if a LFSR has maximum-length then also its mirror has maximal-length.

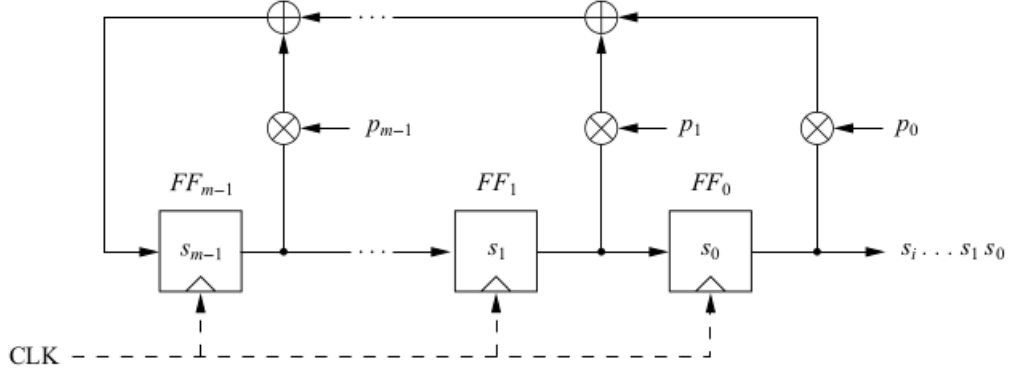


Figure 4: General form of an LFSR of degree m

In the upper figure we have m flip flops and m possible feedback locations, all combined by the XOR operation. Whether a feedback path is active or not, is defined by the feedback coefficient  $p_0, p_1, \dots, p_{m-1}$ . If  $p_i = 1$  (closed switch), the feedback is active. If  $p_i = 0$  (open switch), the corresponding flip-flop output is not used for the feedback. If we multiply the output of flip-flop i by its coefficient  $p_i$ , the result is either the output value if  $p_i = 1$ , which corresponds to a closed switch, or the value zero if  $p_i = 0$ , which corresponds to an open switch. The values of the feedback coefficients are crucial for the output sequence produced by the LFSR.

Let's assume the LFSR is initially loaded with the values  $s_0, \dots, s_{m-1}$ .

In general, the output sequence can be described as:

$$s_{m+i} = \sum_{j=0}^{m-1} s_{i+j} p_j \text{ mod } 2 \quad (6)$$

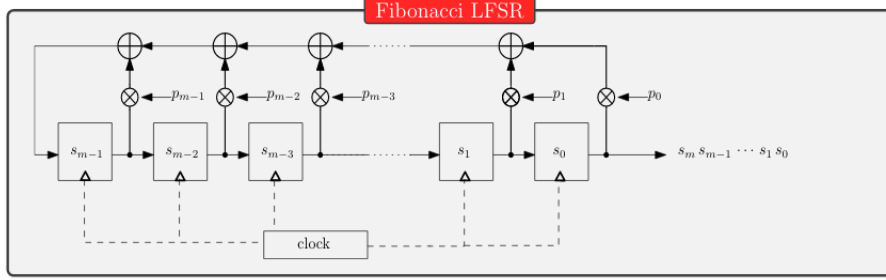
Clearly, the output values are given through a combination of some previous output values. LFSRs are sometimes referred to as linear recurrences.

LFSRs are often specified by polynomials using the following notation: An LFSR with a feedback coefficient vector  $(p_{m-1}, \dots, p_1, p_0)$  is represented by the polynomial

$$P(x) = x^m + p_{m-1}x^{m-1} + \dots + p_1x + p_0 \quad (7)$$



### 3.1 Fibonacci LFSR



The output bit  $s_m$  is computed as a feedback

$$s_m = f(\mathbf{s}) = \sum_{j=0}^{m-1} s_j \cdot p_j \quad (8)$$

The feedback polynomial is  $P(x) = 1 + p_{m-1}x + \dots + p_1x^{m-1} + p_0x^m$  and the characteristic polynomial is  $\chi_L(x) = x^m P(\frac{1}{x}) = p_0 + p_1x + \dots + p_{m-1}x^{m-1} + x^m$ . We can see the  $\mathbf{s}$  states of the LFSR as line vectors  $\mathbf{s} = [s_{m-1} s_{m-2} s_{m-3} \dots s_1 s_0]$  and the transition to the state  $\mathbf{s}'$  is given by the multiplication  $\mathbf{s} \cdot L = \mathbf{s}'$  Where  $L$  is the linear map:

$$L = \begin{bmatrix} p_{m-1} & 1 & 0 & 0 & \dots & 0 \\ p_{m-2} & 0 & 1 & 0 & \dots & 0 \\ p_{m-3} & 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ p_1 & 0 & 0 & 0 & \dots & 1 \\ p_0 & 0 & 0 & 0 & \dots & 0 \end{bmatrix}$$

It's important to notice that after some cycles the LFSR is replicated, then it becomes periodic. A LFSR can't have a period greater than  $2^m - 1$

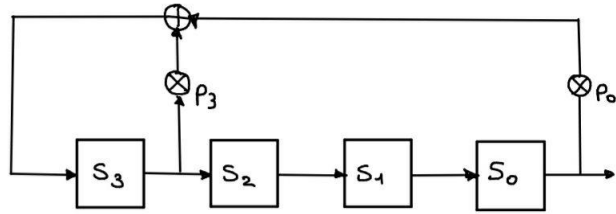
**Example:**

Let's run the Fibonacci LFSR specified by  $\chi_L(x) = x^4 + x^3 + 1$  from the state [1000].  $s_3 = 1, s_2 = 0, s_1 = 0, s_0 = 0$

$$\chi_L(x) = p_0 + p_1x + \dots + p_{m-1}x^{m-1} + x^m$$

So  $p_3 = 1, p_2 = 0, p_1 = 0, p_0 = 1$

The period is 15 ( $2^4 - 1$ ) and The schema is the following one:



$CLK$	$s_3$	$s_2$	$s_1$	$s_0$
0	1	0	0	0
1	1	1	0	0
2	1	1	1	0
3	1	1	1	1
4	0	1	1	1
5	1	0	1	1
6	0	1	0	1
7	1	0	1	0
8	1	1	0	1
9	0	1	1	0
10	0	0	1	1
11	1	0	0	1
12	0	1	0	0
13	0	0	1	0
14	0	0	0	1
15	1	0	0	0

And so on.

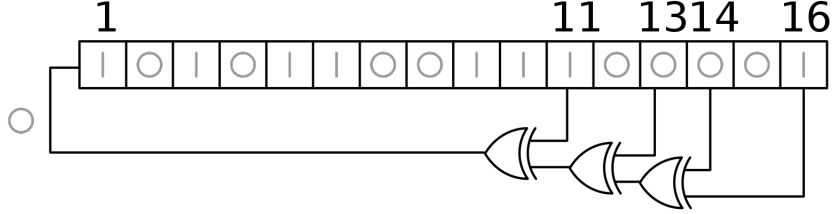


Figure 5: A 16-bit Fibonacci LFSR. The feedback tap numbers shown correspond to a primitive polynomial in the table, so the register cycles through the maximum number of 65535 states excluding the all-zeroes state. If the taps are at the 16th, 14th, 13th and 11th bits (as shown), the feedback polynomial is  $x^{16} + x^{14} + x^{13} + x^{11} + 1$ .

The key stream output  $s_0, s_1, \dots$  determines a formal power series called *generatrix function*

$$\sum_{j=0}^{\infty} s_j x^j = s_0 + s_1 x + s_2 x^2 + \dots \quad (9)$$

It is possible to show that

$$\sum_{j=0}^{\infty} s_j x^j = \frac{Q(x)}{P(x)} \quad (10)$$

Where  $P(x)$  is the feedback polynomial of the LFSR and

$$Q(x) = \sum_{\mu=0}^{\mu=m-1} \left( \sum_{j=m-\mu}^m s_{\mu+j-m} p_j \right) x^{\mu} \quad (11)$$

depends on the initial state of the register.

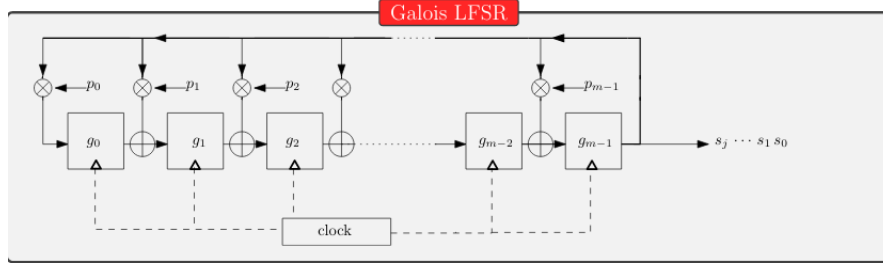
One interesting consequence of the above equation is that it can be used to improve the implementation of key stream by reducing the taps of the LFSR. Here an example: since

$$1 + x + x^8 = (1 + x^2 + x^3 + x^5 + x^6)(1 + x + x^2)$$

the key stream produced by a LFSR with feedback polynomial  $1 + x^2 + x^3 + x^5 + x^6$  can be obtained using a LFSR with feedback polynomial  $1 + x + x^8$

**Exercise:** If a key stream satisfies  $s_{t+6} = s_t + s_{t+1} + s_{t+3} + s_{t+4}$  then it also satisfies  $s_{t+8} = s_t + s_{t+7}$

### 3.2 Galois LFSR



In the Galois configuration the state  $\mathbf{s} = [g_0 g_1 \dots g_{m-2} g_{m-1}]$  is regarded as a polynomial  $s(x) = g_0 + g_1 x + \dots + g_{m-2} x^{m-2} + g_{m-1} x^{m-1}$

When the system is clocked the new state is  $s'(x) = x \oplus s(x)$   
 where  $\oplus$  is as in the Galois cipher with

$$G(s) = \chi_L(x) = x^m + p_{m-1}x^{m-1} + \dots + p_1x + p_0$$

So

$$x \times s(x) = g_0x + g_1x^2 + \dots + g_{m-2}x^{m-1} + g_{m-1}x^m$$

and to get the remainder of the division by  $\chi_L(x)$  we replace  $x^m$  by  $p_{m-1}x^{m-1} + \dots + p_1x + p_0$  and add bits (xor):

$$\begin{array}{r} g_0x \quad \dots \quad g_{m-1}x^{m-1} \\ g_{m-1}p_0 \quad g_{m-1}p_1x \quad g_{m-1}p_2x^2 \quad \dots \quad g_{m-1}p_{m-1}x^{m-1} \\ \hline g_{m-1}p_0 \quad (g_0 + g_{m-1}p_1)x \quad (g_1 + g_{m-1}p_2)x^2 \quad \dots \quad (g_{m-2} + g_{m-1}p_{m-1})x^{m-1} \end{array}$$

Here the linear map L:

$$L = \begin{bmatrix} 0 & \mathbf{1} & 0 & 0 & \dots & 0 \\ 0 & 0 & \mathbf{1} & 0 & \dots & 0 \\ 0 & 0 & 0 & \mathbf{1} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & \mathbf{1} \\ p_0 & p_1 & p_2 & p_3 & \dots & p_{m-1} \end{bmatrix}$$

The passage to a new state  $\mathbf{s}'$  is given by the multiplication  $\mathbf{s} \cdot L = \mathbf{s}'$ . Let's explain it better: In the Galois configuration, when the system is clocked, bits that are not taps are shifted one position to the right unchanged. The taps, on the other hand, are XORed with the output bit before they are stored in the next position. The new output bit is the next input bit. The effect of this is that when the output bit is zero, all the bits in the register shift to the right

unchanged, and the input bit becomes zero. When the output bit is one, the bits in the tap positions all flip (if they are 0, they become 1, and if they are 1, they become 0), and then the entire register is shifted to the right and the input bit becomes 1.

To generate the same output stream, the order of the taps is the counterpart of the order for the conventional LFSR, otherwise the stream will be in reverse. Note that the internal state of the LFSR is not necessarily the same. The Galois register shown has the same output stream as the Fibonacci register in the first section. A time offset exists between the streams, so a different startpoint will be needed to get the same output each cycle. Furthermore the output stream  $\dots s_{m-1} \dots s_2 s_1 s_0$  of a Galois LFSR with  $G(x) = \chi_L(x)$  is the same as the output of a Fibonacci LFSR with  $\chi_L(x)$  and initial state  $s_{m-1} \dots s_2 s_1 s_0$ .

The calculation of  $\mathbf{s} \cdot L$  of the Galois LFSR can be furthermore done using only bit operations. Namely:

$$\mathbf{s} \cdot L = \begin{cases} \text{shiftright}(\mathbf{s}) & \text{if } g_{m-1} = 0 \\ \text{shiftright}(\mathbf{s} \oplus \mathbf{p}) & \text{if } g_{m-1} = 1 \end{cases}$$

where  $\mathbf{p} = [p_0 p_1 p_2 p_3 \dots p_{m-1}]$ .

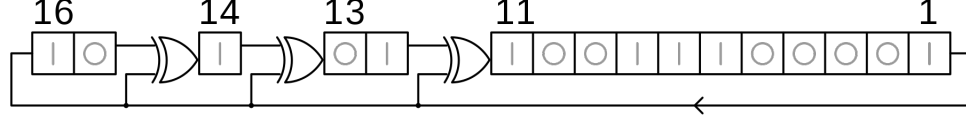


Figure 6: A 16-bit Galois LFSR. The register numbers above correspond to the same primitive polynomial as the Fibonacci example but are counted in reverse to the shifting direction.