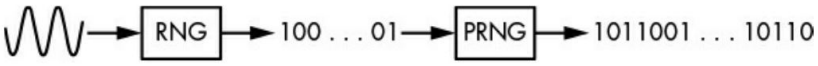
Crypto 2

Note: this material is not intended to replace the live lecture for students.

Contents

2.1 Randomness . . . 2
2.1.1 TRNG . . . 4
2.1.2 PRNG . . . 5
2.1.3 Tests . . . 6
2.2 Operation Modes: Synchronized vs. Unsynchronized . . . 7
2.2.1 Synchronized . . . 7
2.2.2 Unsynchronized . . . 8
2.3 The Lehmer generator and LCG . . . 9
2.3.1 Park Miller RNG . . . 10
2.4 Stream Ciphers: LFSR . . . 11
2.4.1 Fibonacci LFSRs . . . 12
2.4.2 Galois LFSRs . . . 14
2.4.3 Several LFSRs and the Correlation Attack . . . 16
2.5 Stream Ciphers: Permutation . . . 17
2.5.1 Keccak-Sponge key stream . . . 17
2.5.2 The Keccak-f permutations . . . 18
2.6 **Bibliography** . . . 21



2.1 Randomness

What is randomness ?

A pseudo-random sequence is a vague notion embodying the idea of a sequence in which each term is unpredictable to the uninitiated and whose digits pass a certain number of tests traditional with statisticians and depending somewhat on the uses to which the sequence is to be put.

[Lehmer51, pag. 143]

Why 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.

[Aumasson18, Chapter 2]

Figure 2.1.1: Synchronous Stream Cipher

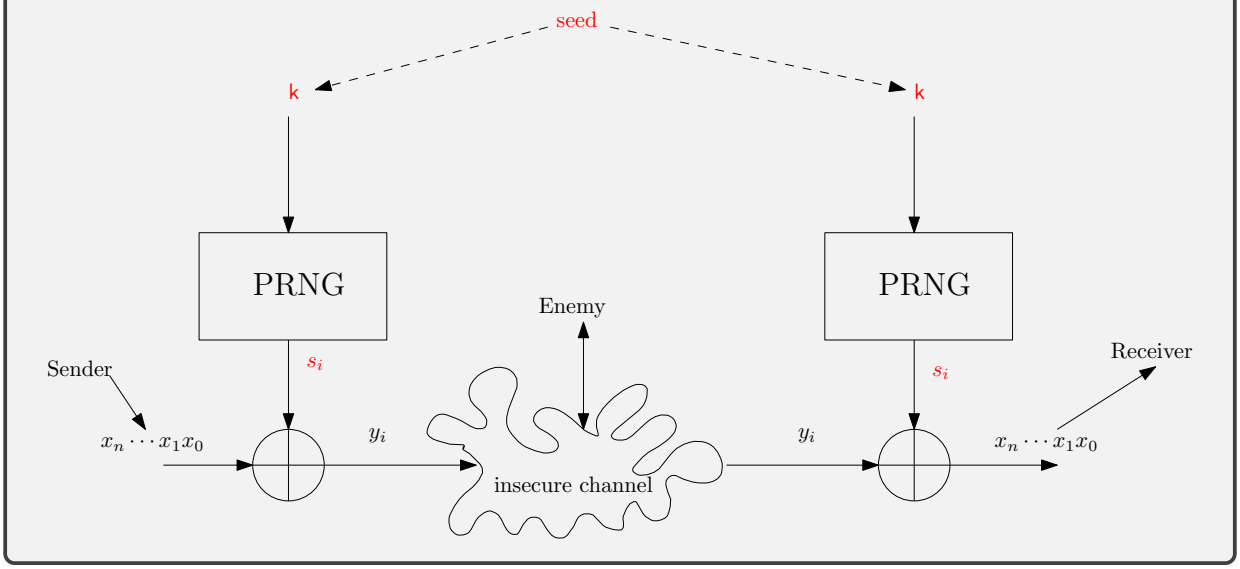


Figure 2.1.2: Probabilistic Encryption

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.

2.1.1 TRNG

> In practice, the cryptoequipment must contain a true random number generator (e.g., a noisy diode) for generating K,

[DH76, pag. 648]

Figure 2.1.3: True Random Number Generator

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.

This SSL Accelerator computer card uses a hardware random number generator to generate cryptographic keys to encrypt data sent over computer networks.

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 attained.

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).

2.1.2 PRNG

2.1.4 PRNG

Un PRNG is a function $G : \mathbb{Z}_2^n \to \mathbb{Z}_2^{l(n)}$, $l(n) > n$ (*expansion factor*) such that no adversary $\mathcal{A}$ succeed with probability $> 1/2$ the following:

The PRG indistinguishability experiment $\mathsf{PRG}_{\mathcal{A},G}(n)$:

(a) A uniform bit $b \in \{0,1\}$ is chosen. If $b = 0$ then choose a uniform $r \in \{0,1\}^{\ell(n)}$; if $b = 1$ then choose a uniform $s \in \{0,1\}^n$ and set $r := G(s)$.

(b) The adversary $\mathcal{A}$ 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.

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

It is usual to construct a PRNG from two algorithms (Init, GetBits):

Init takes as input a seed s and an optional IV and outputs an initial state st_0,

GetBits takes an input state st_i, outputs a bit y and update the state to st_{i+1}.

The states st_i are the states in a big buffer or memory called "entropy pool".

2.1.5 (Init, GetBits) - PRNG $G(s)$

$\mathsf{st}_0 = \mathsf{Init}(s, IV)$
for $i = 1$ to $l(n)$:
  $(y_i, \mathsf{st}_i) = \mathsf{GetBits}(\mathsf{st}_{i-1})$
return $y_1, \cdots, y_{l(n)}$

2.1.3 Tests

Cryptographic vs. Non-Cryptographic PRNGs

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, because they're also concerned with the strength of the underlying *operations* used to deliver well-distributed bits.

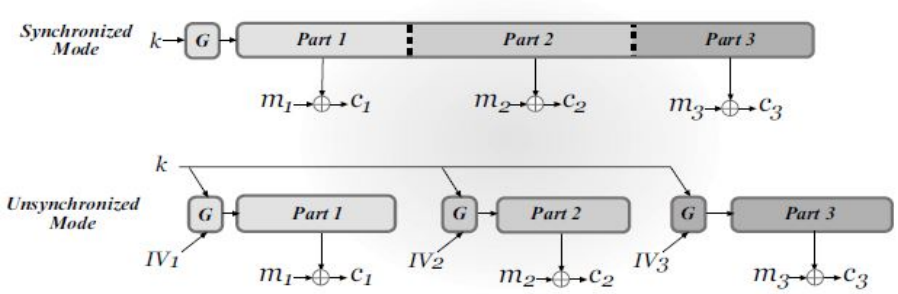
Unfortunately, most PRNGs exposed by programming languages, such as libc's `rand` and `drand48`, PHP's `rand` and `mt_rand`, Python's `random` module, Ruby's `Random` class, and so on, are non-cryptographic. Defaulting to a non-crypto PRNG is a recipe for disaster because it often ends up being used in crypto applications, so be sure to use only crypto PRNGs in crypto applications.

[Aumasson18, Chapter 2]

Will your PRNG do the job?

TestU01

2.2 Operation Modes: Synchronized vs. Unsynchronized



2.2.1 Synchronized

Using $(\mathsf{Init}, \mathsf{GetBits})$ the parties construct $G(\mathsf{k}, 1^l)$ running l-times using a shared k. Concretely both begin by computing $\mathsf{st}_0 = \mathsf{Init}(\mathsf{k})$.
To encrypt the first message m_1 of length l_1 the sender runs $\mathsf{GetBits}$ l_1-times beginning at st_0 getting bits $y_1 \cdots y_{l_1} = \mathsf{pad}_1$ along with an updated state st_{l_1}. It sends $c_1 = \mathsf{Enc}_{\mathsf{k}}(m_1) = \mathsf{pad}_1 \oplus m_1 = \mathsf{pad}_1 \oplus m_1$.

Once the other party receive c_1 it runs $\mathsf{GetBits}$ l_1-times beginning at st_0 getting bits $y_1 \cdots y_{l_1} = \mathsf{pad}_1$ along with an updated state st_{l_1}. It uses pad_1 to recover $m_1 = c_1 \oplus \mathsf{pad}_1$.
Later the sender to encrypt a second message m_2 will run $\mathsf{GetBits}$ at the state st_{l_1} to obtain a second $\mathsf{pad}_2 = y_{l_1+1} \cdots y_{l_1+l_2}$ and an updated state $\mathsf{st}_{l_1+l_2}$ and so on.

NOTE 2.2.1

We remark that in this mode, the stream cipher does not need to use the initialization IV.

2.2.2 Unsynchronized

In this case ciphertexts are pairs :

$$\mathsf{Enc}_{\mathsf{k}}(m_j) = \langle IV_j \, , \, G(\mathsf{k}, IV_j, 1^{|m_j|}) \oplus m_j \rangle$$

where $G(\mathsf{k}, IV_j, 1^{|m|})$ is the pad obtained by running $\mathsf{Init}(\mathsf{k}, IV_j)$. Decryption is performed in the natural way.

NOTE 2.2.2

Tipical mistakes are related to the optional IV e.g. a fixed constant in soft or hardware. If IV is random then $\mathsf{Enc}_{\mathsf{k}}(m) = \langle IV \, , \, G(\mathsf{k}, IV, 1^{|m|}) \oplus m \rangle$ is CPA-secure. But not CCA-secure.

Exercise 2.2.3

Let $G(\mathsf{k}, IV, 1^{|m|})$ 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:

$$\mathsf{Enc}_{\mathsf{k}}(m) = \langle IV, G(\mathsf{k}, IV, 1^{|m|}) \oplus m \rangle$$

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

$$\mathsf{Dec}_{\mathsf{k}}(\langle IV, c \rangle) = G(\mathsf{k}, IV, 1^{|c|}) \oplus c = m$$

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.

2.3 The Lehmer generator and LCG

METHODS IN LARGE-SCALE UNITS

of this session—the Monte Carlo method. In this method it is necessary to produce random variables. The problem here is not one of producing a table of random digits to be published and used by others. On the contrary, one can think ideally of a perfect stream of these random numbers produced at high speed by the machine and passing by a "gate." Whenever the computer needs a number it opens the gate and takes one. More explicitly, we might list the following desiderata:

(1) An unlimited sequence of randomly arranged eight-digit numbers;

(2) A simple process by which the machine may produce the sequence:

$$u_n = f(u_{n-1}, u_{n-2}, \cdots, u_{n-k});$$

(3) Immediate access by the machine to the current number of the sequence whenever necessary; of course, the whole sequence need not be retained in the machine.

If we examine these desiderata we see at once that they are inconsistent. In the first place, the number of eight-digit numbers is not unlimited. There are in fact only 100 million of them. Secondly, condition (2) forces the sequence to be ultimately periodic and therefore not random. We therefore scale down our demands and modify (1) and (2) to read:

(1) Millions of pseudo-random numbers u_n;

(2) $u_n = f(u_{n-1})$, f a simple function.

A pseudo-random sequence is a vague notion embodying the idea of a sequence in which each term is unpredictable to the uninitiated and whose digits pass a certain number of tests traditional with statisticians and depending somewhat on the uses to which the sequence is to be put. The worst possible departure from randomness is to have the period of the sequence small or equal to one. In constructing the function f, therefore, it is of the utmost importance to obtain one that produces a guaranteed proper period of immense length.

[Lehmer51]

Here is the general form a of LCG:

$$\begin{aligned} s_0 &= \text{seed} \\ s_{i+1} &\equiv a \cdot s_i + b \pmod{m},\ i = 0, 1, \cdots \end{aligned}$$

A well-known example is *rand()* of ANSI C:

$$\begin{aligned} s_0 &= 12345 \\ s_{i+1} &\equiv 1103515245 \cdot s_i + 12345 \pmod{2^{31}},\ i = 0, 1, \cdots \end{aligned}$$

Exercise 2.3.1

Setting $b = 0$, $s_0 = 47594118$, $a = 23$ e $m = 10^8 + 1$ show that each s_n is divisible by 17. It is possible to shat 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* .

2.3.1 Park Miller RNG

2.3.2 Park Miller RNG

```
# Park - Miller RNG
#
# https://www.youtube.com/watch?v=EXsLyfqwopg
#
# http://www.firstpr.com.au/dsp/rand31/p1192-park.pdf

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
```

2.4 Stream Ciphers: LFSR

A linear-feedback shift register (LFSR) is a shift register whose state transition is linear.

The state $\mathbf{s}$ is a row vector of m bits, i.e. $\mathbf{s} \in \{0,1\}^m$.

The sequence of bits in the rightmost position of the state is **the output stream** or key stream.

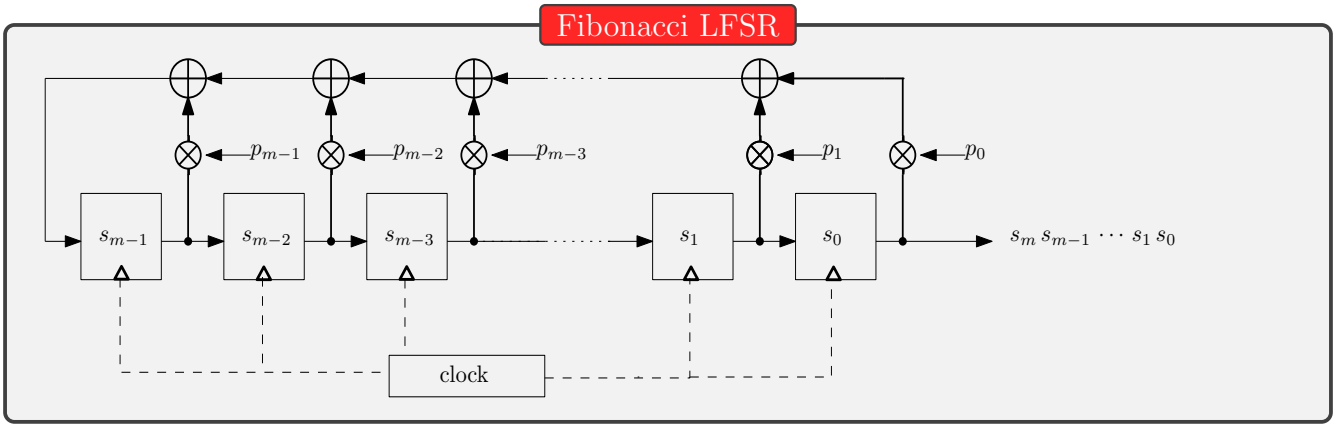
The bits in the LFSR state that influence its behaviour are called **taps**.

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.

Here I discuss: Fibonacci and Galois LFSRs. In both cases the LFSR is specified by either the characteristic polynomial $\chi_{\mathrm{L}}(x)$ of the linear transition map or by the so called **feedback polynomial** $P(x)$ which is the reciprocal of $\chi_{\mathrm{L}}(x)$.

Two given LFSRs are **mirror** LFSRs if they 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 also its mirror has maximal-length.

2.4.1 Fibonacci LFSRs



The output bit s_m is computed as a feedback

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

The feedback polynomial is : $P(x) = 1 + p_{m-1}x + \cdots + p_1 x^{m-1} + p_0 x^m$ and the characteristical polynomial is

$$\chi_{\mathrm{L}}(x) = x^m P(\frac{1}{x}) = p_0 + p_1 x + \cdots + p_{m-1}x^{m-1} + x^m \ .$$

Here the involved linear map L is:

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

Being the state $\mathbf{s} = [s_{m-1}s_{m-2}s_{m-3}\cdots s_1 s_0]$ the transition to the state $\mathbf{s}'$ is given by the multiplication:

$$\mathbf{s} \cdot \mathrm{L} = \mathbf{s}' \,,$$

or at a bits level:

$$\mathbf{s}' = [f(\mathbf{s}) s_{m-1}s_{m-2}s_{m-3}\cdots s_2 s_1] \,.$$

2.4.1 Example

Run the Fibonacci LFSR specified by $\chi_{\mathrm{L}}(x) = x^4 + x^3 + 1$ from the state [1000]. How long is its period? Then run its mirror and check that it cycle through the output stream in reverse order.

The generatrix power series

The key stream output $s_0, s_1, \cdots$ determines a formal power series[a]:

$$\sum_{j=0}^{\infty} s_j x^j = s_0 + s_1 x + s_2 x^2 + \cdots$$

It is possible to show that

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

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

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

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

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

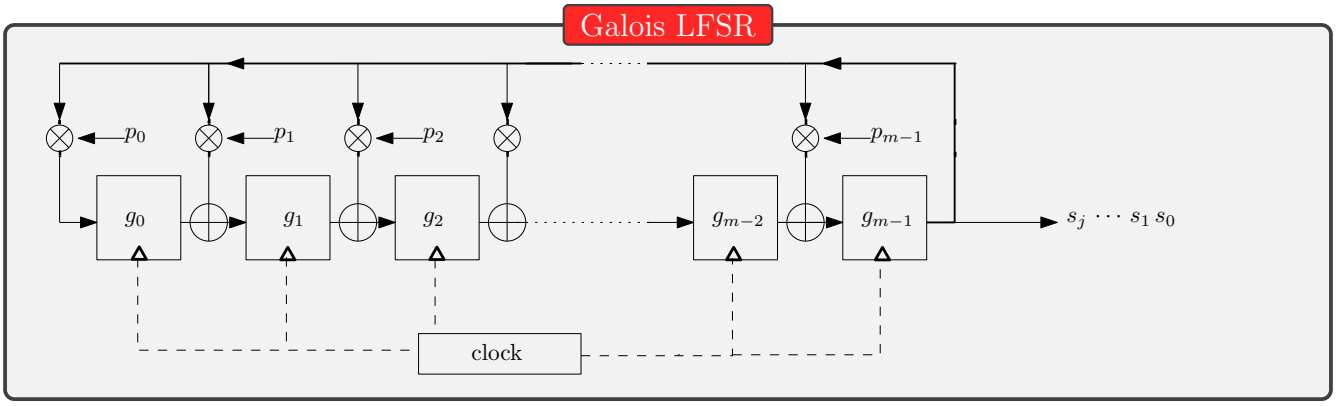
the key stream produce by a LFSR with feedback polynomial $1 + x^2 + x^3 + x^5 + x^6$ can be obtained using a LFSR with feedback polynomial $1 + x + x^8$.

[a] called generatrix function

Exercise 2.4.2

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

2.4.2 Galois LFSRs



In the Galois configuration the state $\mathbf{s} = [g_0 g_1 \cdots g_{m-2} g_{m-1}]$ is regarded as a polynomial

$$\mathrm{s}(x) = g_0 + g_1 x + \cdots + g_{m-2} x^{m-2} + g_{m-1} x^{m-1}$$

When the system is clocked the new state is

$$\mathrm{s}'(x) = x \otimes \mathrm{s}(x)$$

where $\otimes$ is as in the Galois cipher with $G(x) = \chi_{\mathrm{L}}(x) = x^m + p_{m-1}x^{m-1} + \cdots + p_1 x + p_0$. So

$$x \times \mathrm{s}(x) = g_0 x + g_1 x^2 + \cdots + g_{m-2} x^{m-1} + g_{m-1} x^m$$

and to get the remainder of the divison by $\chi_{\mathrm{L}}(x)$ we replace x^m by $p_{m-1}x^{m-1} + \cdots + p_1 x + p_0$ and add bits (xor):

$$\begin{array}{llll} & g_0 x & g_1 x^2 & \cdots & g_{m-2} x^{m-1} \\ g_{m-1}p_0 & g_{m-1}p_1 x & g_{m-1}p_2 x^2 & \cdots & g_{m-1}p_{m-1} x^{m-1} \\ \hline g_{m-1}p_0 & (g_0 + g_{m-1}p_1)x & (g_1 + g_{m-1}p_2)x^2 & \cdots & (g_{m-2} + g_{m-1}p_{m-1})x^{m-1} \end{array}$$

Here the linear map L:

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

the passage to a new state $\mathbf{s}'$ is given by the multiplication:

$$\mathbf{s} \cdot \mathrm{L} = \mathbf{s}'$$

Exercise 2.4.3

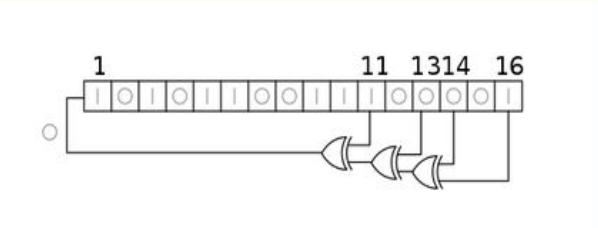
Show that the calculation of $\mathbf{s} \cdot \mathrm{L}$ of the Galois LFSR can be done using only bit operations. Namely, show that

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

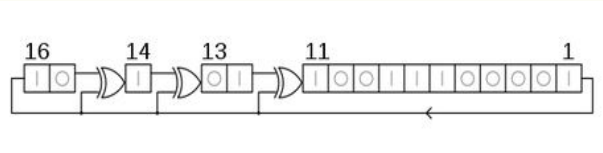
where $\mathbf{p} = [p_0 p_1 p_2 p_3 \cdots p_{m-1}]$.

Exercise 2.4.4

Write the matrices L and their $\chi_{\mathrm{L}}(x)$ for the two examples of wikipedia. Namely:



and



Galois & Fibonacci

The output stream $\cdots s_{m-1} \cdots s_2 s_1 s_0$ of a Galois LFSR with $G(x) = \chi_{\mathrm{L}}(x)$ is the same as the output stream of a Fibonacci LFSR with $\chi_{\mathrm{L}}(x)$ and initial state $s_{m-1} \cdots s_2 s_1 s_0$.

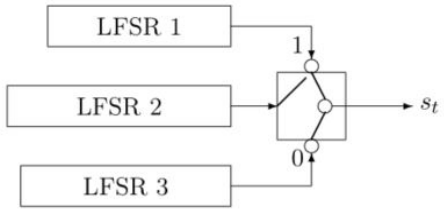
2.4.3 Several LFSRs and the Correlation Attack

Geffe Generator

This PRNG is composed of three LFSRs of distinct lenghts combined by the boolean function

$$f(x_1, x_2, x_3) = x_1x_2 + x_2x_3 + x_3 \ .$$

It is worth noticing that this function corresponds to an IF: if $x_2 = 0$, then $f(x_1, x_2, x_3) = x_3$ and if $x_2 = 1$, $f(x_1, x_2, x_3) = x_1$.



Video by Anne Canteaut about Boolean Functions
https://en.wikipedia.org/wiki/Correlation_attack

NOTE 2.4.5

There are several PRNG used in crypto constructed by using shift registers e.g. Mersenne Twister, Trivium, E0, A5/1.

2.5 Stream Ciphers: Permutation

The oracle model of a Permutation.

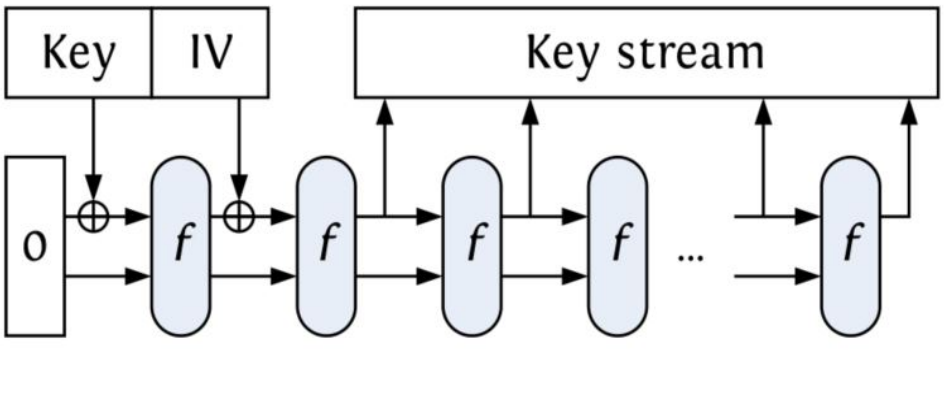
ORACLES. For convenience, a random oracle R is a map from $\{0,1\}^*$ to $\{0,1\}^\infty$ chosen by selecting each bit of $R(x)$ uniformly and independently, for every x. Of course no actual protocol uses an infinitely long output, this just saves us from having to say how long "sufficiently long" is. We denote by 2^∞ the set of all random oracles.

[BeRo95, pag. 6]

My function behaves as a random oracle, or it cannot be distinguished form a random permutation

2.5.1 Keccak-Sponge key stream

2.5.1 Keccak-sponge stream cipher

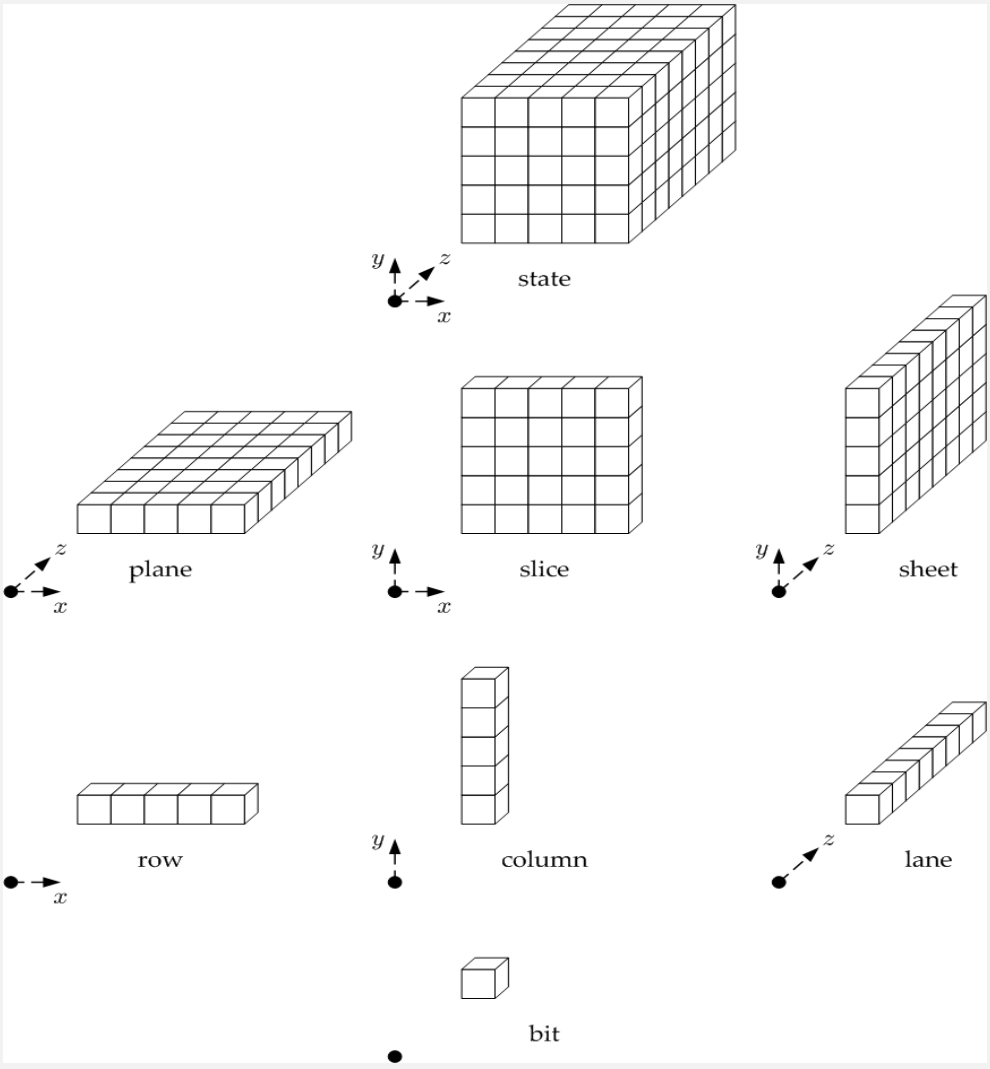


2.5.2 The Keccak-f permutations

There are 7 Keccak-f permutations of $\{0,1\}^b$, indicated Keccak-f[b] , where $b = 25 \times 2^l$ and l ranges from 0 to 6.

The elements of $\{0,1\}^b$ are regarded as a 3-dimensional array:

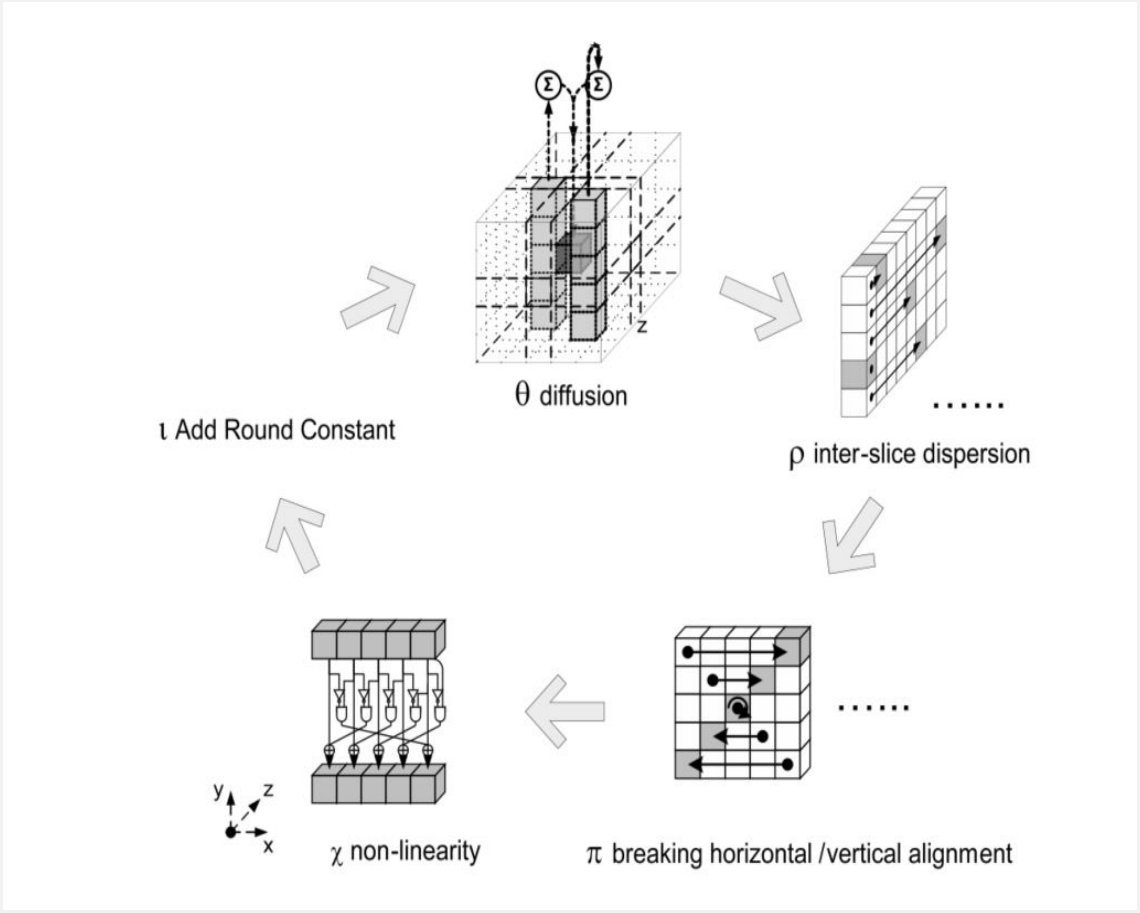
Figure 2.5.2: Keccak-f-State Pieces



To compute Keccak-f[b](x) there are $12 + 2l$ rounds, e.g. Keccak-f[25] has 12 rounds, Keccak-f[1600] has 24 rounds.

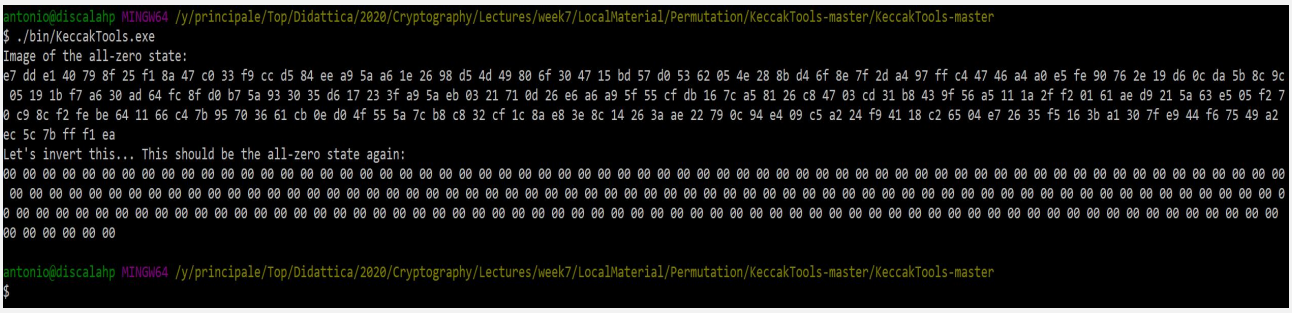
A round consists of 5 invertible steps mappings:

Figure 2.5.3: Keccak-f Round



https://keccak.team/index.html

Figure 2.5.4: Keccak-f[1600](0)



2.6 Bibliography

Here is the list of books I used for the preparation of this note.

[Aumasson18] Jean-Philippe Aumasson, *Serious Cryptography: A Practical Introduction to Modern Encryption*, No Starch Press, 2018.

[KatLin15] Jonathan Katz; Yehuda Lindell, *Introduction to Modern Cryptography* Second Edition, Chapman & Hall/CRC, Taylor & Francis Group, 2015.

[Paar10] Paar, Christof, Pelzl, Jan, *Understanding Cryptography, A Textbook for Students and Practitioners*, Springer-Verlag, 2010.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling , Brian P. Flannery *Numerical Recipes in C The Art of Scientific Computing*, Second Edition, CAMBRIDGE UNIVERSITY PRESS, 1992.

Here is the list of papers:

[BeRo95] Bellare, Mihir and Rogaway, Phillip; *Random Oracles are Practical: A Paradigm for Designing Efficient Protocols*, Proceedings of the First Annual Conference on Computer and Communications Security https://cseweb.ucsd.edu/~mihir/papers/ro.pdf

[BDPA11] Bertoni,G.;Daemen,J.;Peeters,M. and Van Assche, G.: *The Keccak reference*, https://keccak.team/files/Keccak-reference-3.0.pdf

[BDPA11b] Bertoni,G.;Daemen,J.;Peeters,M. and Van Assche, G.: *Cryptographic sponge functions*, https://keccak.team/files/CSF-0.1.pdf

[DH76] Diffie, W.; Hellman, M. *New directions in cryptography*, (1976). IEEE Transactions on Information Theory. 22 (6): 644654.

[GM82] Goldwasser, S. and Micali, S.; *Probabilistic Encryption & How To Play Mental Poker Keeping Secret All Partial Information*, Proc. 14th Symposium on Theory of Computing: 365–377. (1982)

[Lehmer51] Lehmer, D.H.; *Mathematical methods in large-scale computing units.*, Proceedings of a Second Symposium on Large-Scale Digital Calculating Machinery, 1949, pp. 141–146. Harvard University Press, Cambridge, Mass., 1951.

[Muk13] Pratyay Mukherjee; *An Overview of eSTREAM Ciphers*, Centre of Excellence in Cryptology, Indian Statistical Institute, Kolkata, India, (2013). http://www.cs.au.dk/~pratyay/eSTREAM.pdf

and some interesting links:

NIST sp800-22 r1a

https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch37.html

https://nullprogram.com/blog/2019/04/30/