

# Cryptography and PNRG

---



# What we are going to see:

- Introduction to exploits
  - pwntools
  - XOR function
- Cryptography
  - Common use case
  - What is Random?

What are we going to use:

- python: **pwntools**, **z3-solver**
- platform [workshop.m0lecon.it](http://workshop.m0lecon.it)

# Writing exploits in Python

---

# What is an exploit?

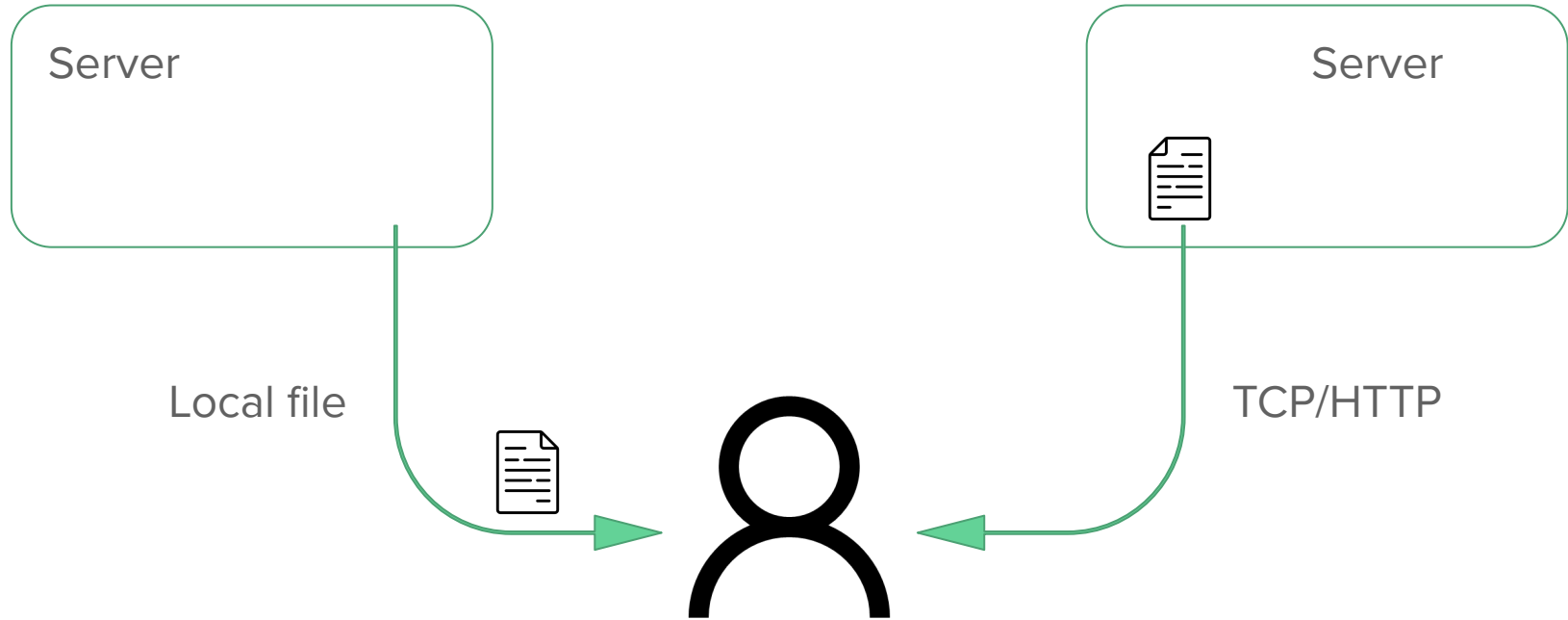
An **exploit** is a “a **piece of code** or a set of commands that takes advantage of a **vulnerability**, or weakness, in a computer system, software, or hardware for a **malicious purpose**, such as gaining unauthorized access or stealing data”

Usually it interacts with the target software through the same interface as normal users

## 3 main scenarios:

- CLI interface
- HTTP API
- In-browser interaction

# Common scenario



For HTTP we have **requests**

How to handle TCP connection? what about local exploits?

# Interacting: CLI interface

- Local interaction: **stdin** and **stdout**
- Remote processes: **nc** to simulate a terminal over TCP


How to handle both types? **pwntools**

python package with lots of utilities to write exploits

- Supports both local and remote programs, using the same functions
- Makes writing exploits easy, from simple in/out interactions to complex binary exploitation techniques
- Really, really a lot more...

**pip3 install pwntools**

# Using pwntools



```
1  from pwn import *
2
3  conn = process(["python", "my_chall.py"])
4  conn = remote("software-17.challs.olicityber.it", 13000)
5
6  conn.recvline()          # receive single line
7  conn.recvuntil(b'> ')   # receive until prompt
8  conn.recvall()           # receive all data until EOF
9  conn.recv(128)           # receive 128 bytes
10
11 conn.send(b'Hello\n')    # send bytes
12 conn.sendline(b'World!') # send data with newline
13 conn.sendafter(b'> ', b'Input\n') # send data after receiving prompt
14
15 conn.interactive()       # interact with the process and get a shell
```

# XOR

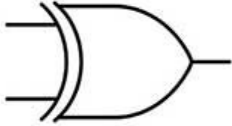
- Definition of the XOR operator
- The role of XOR in cryptography
- Why XOR?
- XOR Properties
- One time pad: how to encrypt with XOR



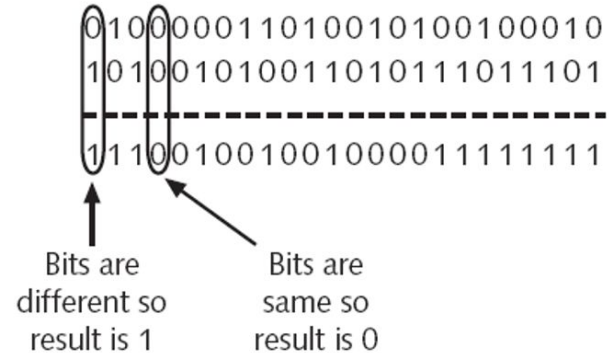
# Definition of the XOR operator

- XOR takes two inputs and returns an output
- It is a bitwise operation, which means each bit of the two inputs is processed separately, producing one bit of output, then the different outputs are concatenated, producing the final output

XOR



A	B	Output
0	0	0
1	0	1
0	1	1
1	1	0



# The role of XOR in cryptography

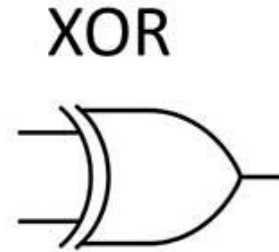
Some examples of cryptographic primitives which rely on the XOR operation:

- Hash functions (sha2, sha3...)
- Symmetric key encryption / decryption
  - Block ciphers (AES-CBC...)
  - Stream ciphers (AES-CTR, ChaCha20...)

...and many more!

# XOR properties

- $a \oplus (b \oplus c) = (a \oplus b) \oplus c$
- $a \oplus b = b \oplus a$
- $a \oplus a = 0$
- $a \oplus 0 = a$
- $a \oplus b \oplus a = b$ 
  - $a \oplus b \oplus a =$
  - $a \oplus a \oplus b =$
  - $0 \oplus b =$
  - $b$










A	B	Output
0	0	0
1	0	1
0	1	1
1	1	0

# Why XOR?

For a given plaintext bit (be it 0 or 1), the output is equally likely to be 0 or 1. So the ciphertext alone holds no information about the plaintext. This doesn't hold for other operators

Example: suppose we were using AND operator to encrypt a message, if a bit in the ciphertext is 1 we know for sure that the corresponding bit in the plaintext is 1 too.

Input		Output (Q)						
								
A	B	AND	OR	INH	XOR	NAND	NOR	XNOR
0	0	0	0	0	0	1	1	1
0	1	0	1	0	1	1	0	0
1	0	0	1	1	1	1	0	0
1	1	1	1	0	0	0	0	1

# One Time Pad: how to “hide” with XOR

Given a message  $p$ , we can make it unreadable using another message  $k$  with the same length in the following way:

$$c = p \oplus k$$

Since  $a \oplus b \oplus a = b$ , the original message can be obtained as follows:

$$c \oplus k = p \oplus k \oplus k = p$$

**Why do we need the key and the plaintext to be the same size?**

# Challenge time

# Cryptography

---

# What is cryptography

From Ancient Greek: κρυπτός, romanized: kryptós "hidden, secret"; and γράφειν graphein, "to write", or -λογία -logia, "study"

“Cryptography is the practice and study of **techniques** for **secure communication** in the presence of adversarial behavior”

Common uses:

- Internet communications (SSL, HTTPS...) and Message applications (e.g. WhatsApp)
- Mobile networks (e.g. GSM)
- Legal documentations (digital signatures)
- Credit-card transactions over Internet
- Blockchains
- ... many more!



# Security principles



But also

- Authentication
- Authorization
- Non-repudiation
- Access control
- many more...

# The tool that is Cryptography

## Confidentiality

### Encryption

- symmetric & asymmetric ciphers
- block & stream ciphers

## Integrity

### Anti-tamper controls

- hash functions

## Authentication

### Digital signatures

HMAC

A cryptographic system is usually made up of many fundamental cryptographic algorithms called **primitives**.

# Something is secret

## Kerchoff's principle:

“The cryptographic key should be the only secret: it would be foolish to rely on our enemies not to discover what algorithms we use because they most likely will. Instead, let's be open about them.”

## Rule: Cryptographic Algorithm Maturity Requirement

A cryptographic algorithm must undergo a minimum of **10 years of public study** and analysis by the global community of crypto-mathematicians before being considered for practical application or standardization. This period ensures that the algorithm has been rigorously tested for vulnerabilities, cryptanalytic resistance, and long-term reliability.

# Pseudo Random Number Generation

---

# PNRG

Is an **algorithm** for generating a sequence of numbers whose properties **approximate the properties** of sequences of random numbers.

The PRNG-generated sequence is not truly random, because it is completely determined by an initial value, called the PRNG's **seed** (which may include truly random values).

Advantages:

- speed
- reproducibility

# PNRG

We will focus on Generators based on linear recurrences:

- LCG
- LFSR
- Mersenne Twister

LCG

---

# Linear Congruential Generator

- One of the oldest PRNG
- Uses a linear function modulo  $m$

$$s_{n+1} = a * s_n + c \text{ mod } m$$

- $m$ ,  $0 < m$  - (prime) modulus
- $a$ ,  $0 < a < m$  - multiplier
- $c$ ,  $0 \leq c < m$  - increment
- $s_0$ ,  $0 \leq s_0 < m$  - seed



# LCG - example

$m = 4294967295$

$a = 1689429238$

$c = 216342301$

$s_0 = 1260873289$

1. seed = 1260873289
2. apply linear transformation:  $a * s_0 + c = 2130156200066166083$
3. transform using the modulo operation = 1117389578
4. repeat:
  - 3590133225
  - 1682940271
  - 930641624
  - 2664134298
  - 3656255800
  - ...

# LCG - example

$m = 2147483648$   
 $a = 1689429238$   
 $c = 216342301$   
 $s_0 = 1260873289$

It's possible to recover the parameters knowing the sequence.

All the operation must be done “modulus  $m$ ”. Practically it means:

- summation, subtraction and multiplications are done normally and before apply the modulus
- the division is replaced by the inverse operation, in python computed as:

```
1 res = pow(a, -1, m)
```

In a more formal way, we are operating within the modular arithmetic system (the ring  $\mathbb{Z}_m$ ), where addition and multiplication are defined modulo  $m$ , and multiplicative inverses exist only for numbers coprime with  $m$ .

# LCG - how to break!

$m = 2147483648$

$a = 1689429238$

$c = 216342301$

$s_0 = 1260873289$

Knowing how to operate modulo  $m$ , recovering  $s_0$ ,  $a$  and  $c$  became simple algebra

Recover **s**:

$$s_0 = (s_1 - c) a^{-1} \bmod m$$

Recover **c**:

$$c = a * s_0 - s_1 \bmod m$$

Recover **a**:

\*\*\*\*\*

# Challenge time

# LFSR

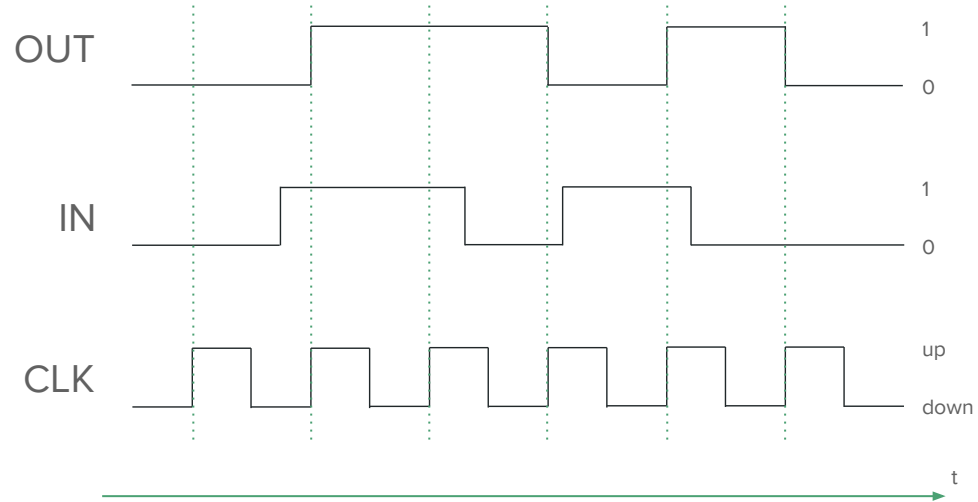
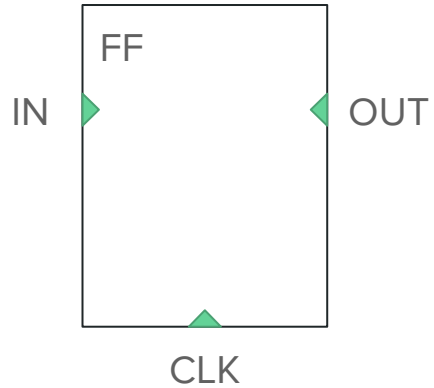
---

# LFSR

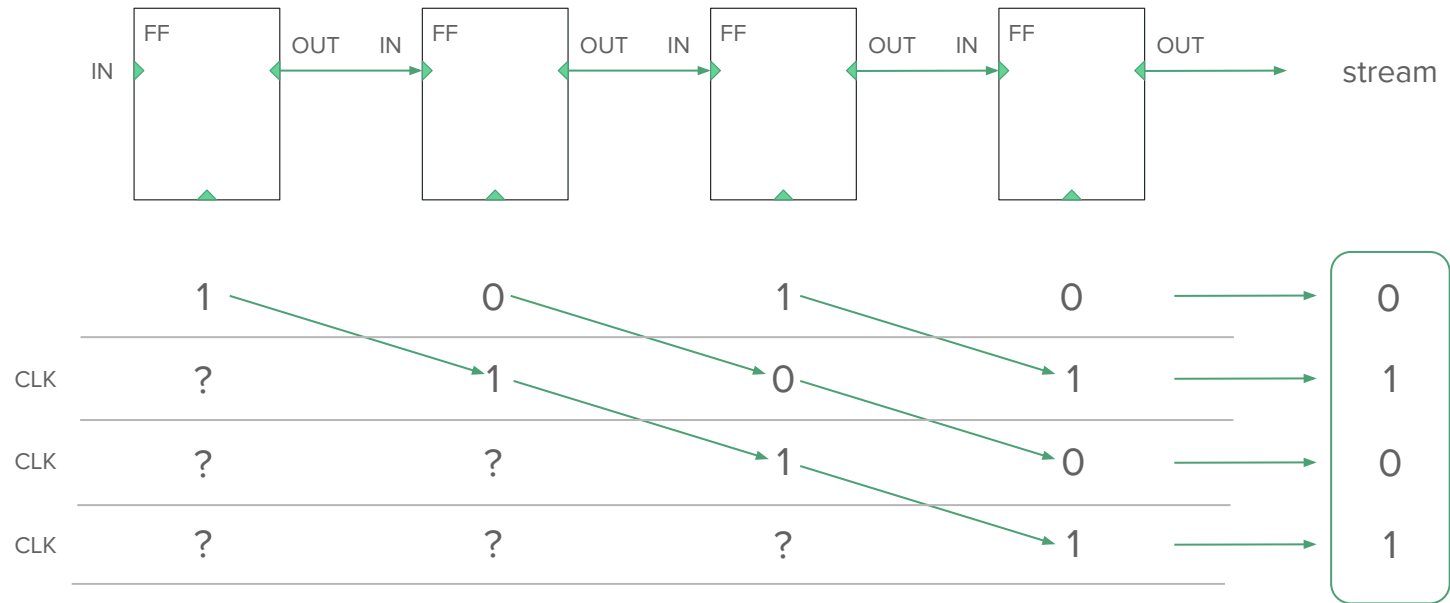
- Linear
- **F**eedback
- **S**hift
- **R**egister

In particular, we will focus on the **Fibonacci** LFSR

# LFSR - Register

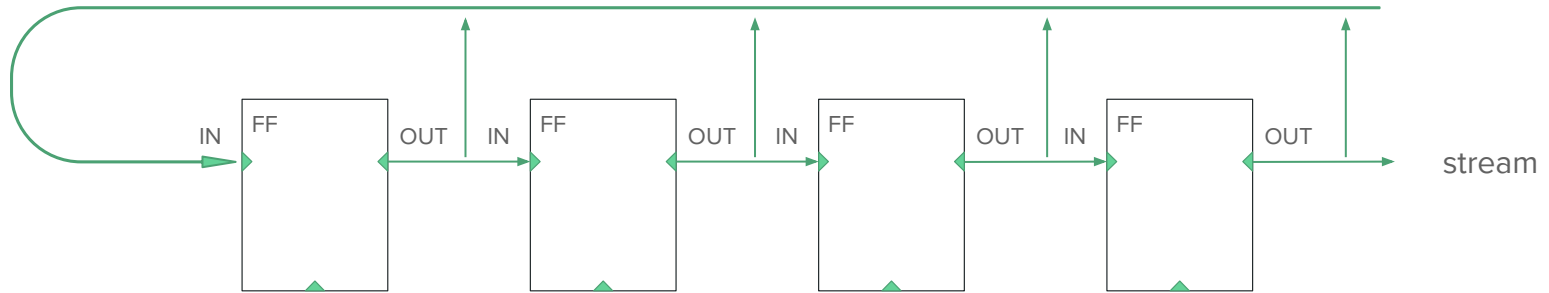


# LFSR - Shift

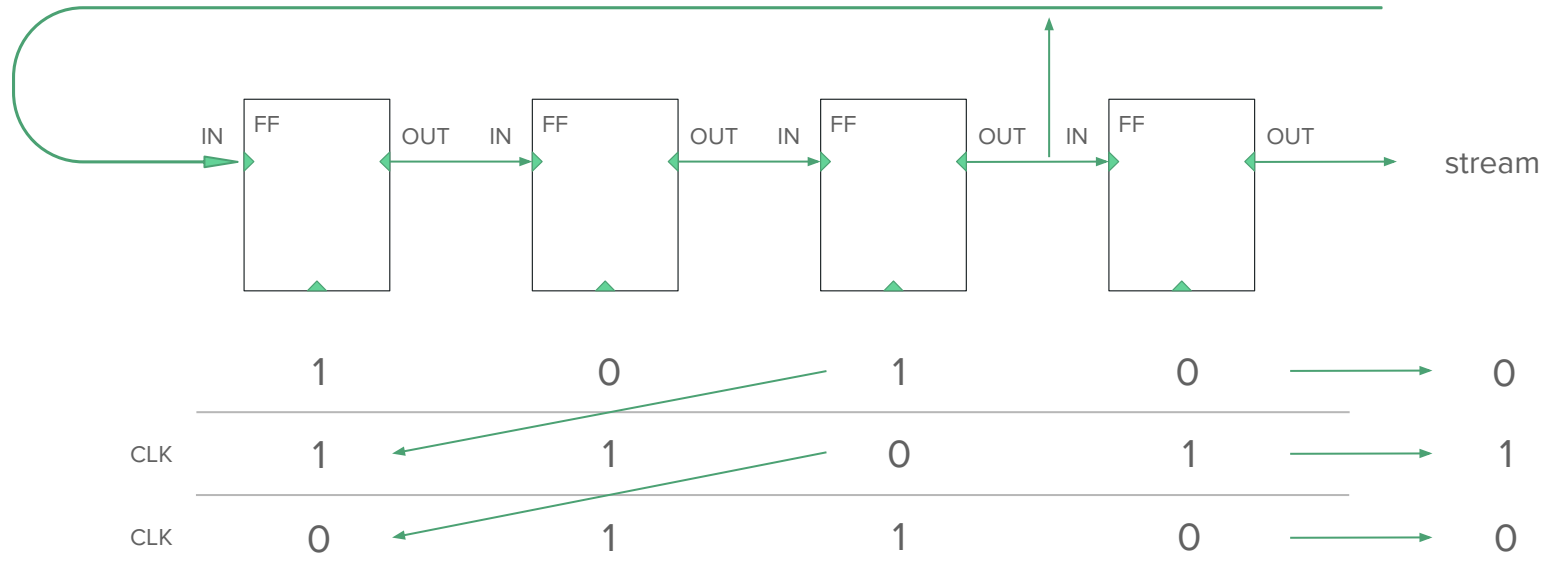




# LFSR - Feedback



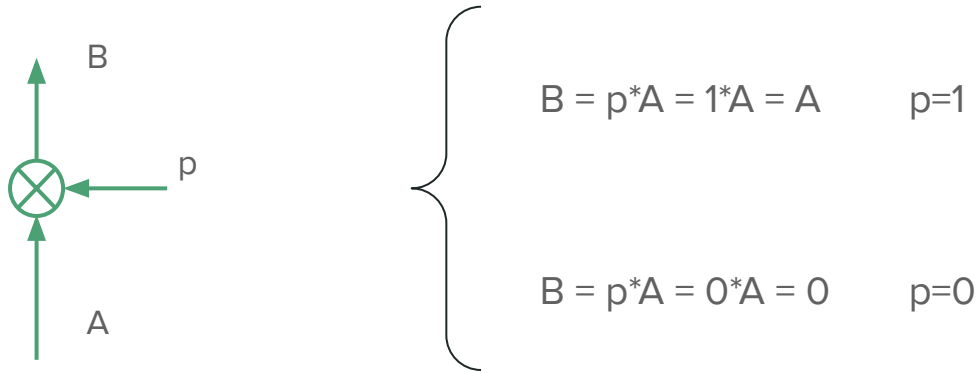
# LFSR - Feedback



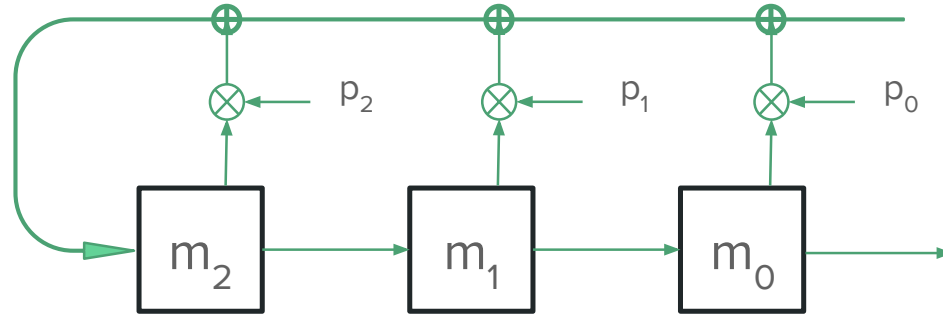
# LFSR - Linear

We are working with a single bit, a simple linear operation is the **xor** that can be viewed as the **sum over 1 bit**.

Each “feedback connection” can be viewed as a weighted gate

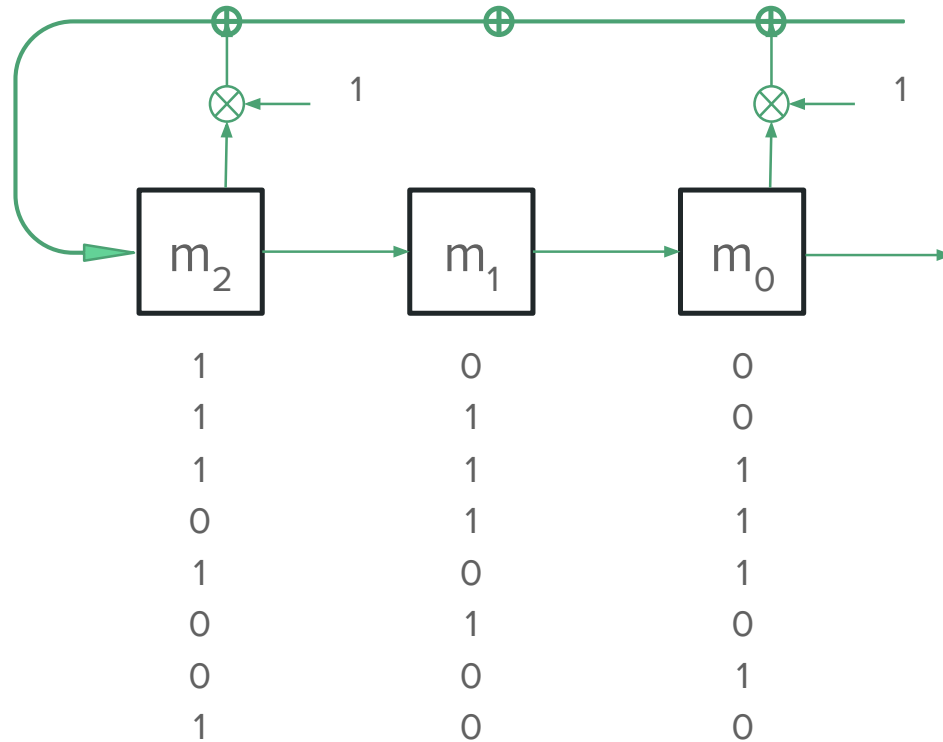


# LFSR - Linear



$$\begin{cases} s_3 = s_2 \otimes p_2 \oplus s_1 \otimes p_1 \oplus s_0 \otimes p_0 \\ s_4 = s_3 \otimes p_2 \oplus s_2 \otimes p_1 \oplus s_1 \otimes p_0 \\ s_5 = s_4 \otimes p_2 \oplus s_3 \otimes p_1 \oplus s_2 \otimes p_0 \end{cases}$$

# LFSR - Example



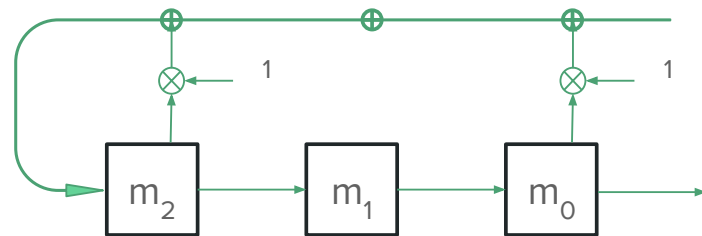
# LFSR - Example

Taps: [0, 2]

Mathematically:  $p_2=1$ ;  $p_1=0$ ;  $p_0=1$

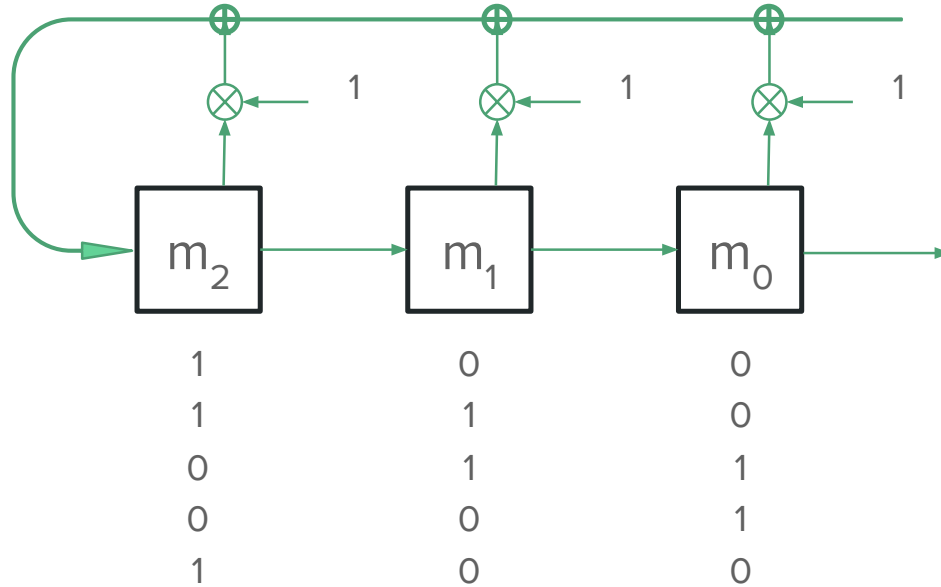
state			
1	0	0	$s_0$
1	1	0	$s_1$
1	1	1	$s_2$
0	1	1	$s_3 = s_2 \otimes 1 + s_1 \otimes 0 + s_0 \otimes 1 = s_2 + s_0 = 1+0$
1	0	1	$s_4 = s_3 + s_1 = 1 + 0 = 1$
0	1	0	$s_5 = s_4 + s_2 = 1 + 1 = 2 = 0 \pmod{2}$
0	0	1	$s_6 = s_5 + s_3 = 0 + 1 = 1$
1	0	0	$s_7 = s_6 + s_4 = 1 + 1 = 0$

seed

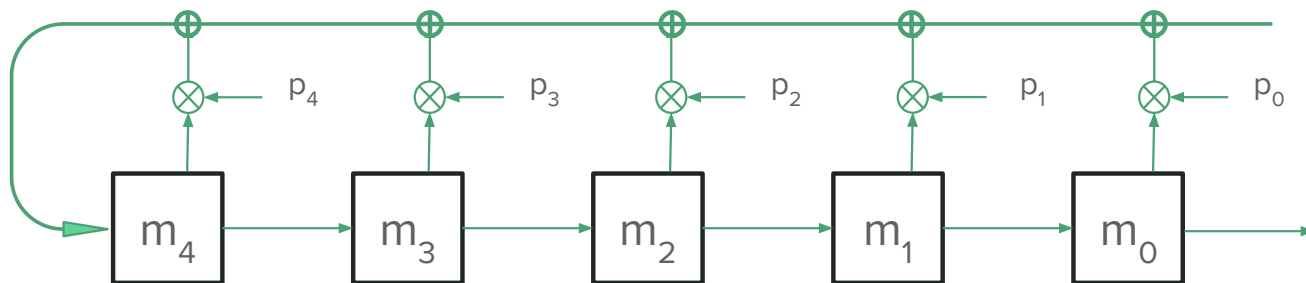


$$\begin{cases} s_3 = s_2 \otimes p_2 \oplus s_1 \otimes p_1 \oplus s_0 \otimes p_0 \\ s_4 = s_3 \otimes p_2 \oplus s_2 \otimes p_1 \oplus s_1 \otimes p_0 \\ s_5 = s_4 \otimes p_2 \oplus s_3 \otimes p_1 \oplus s_2 \otimes p_0 \end{cases}$$

## LFSR - Example 2



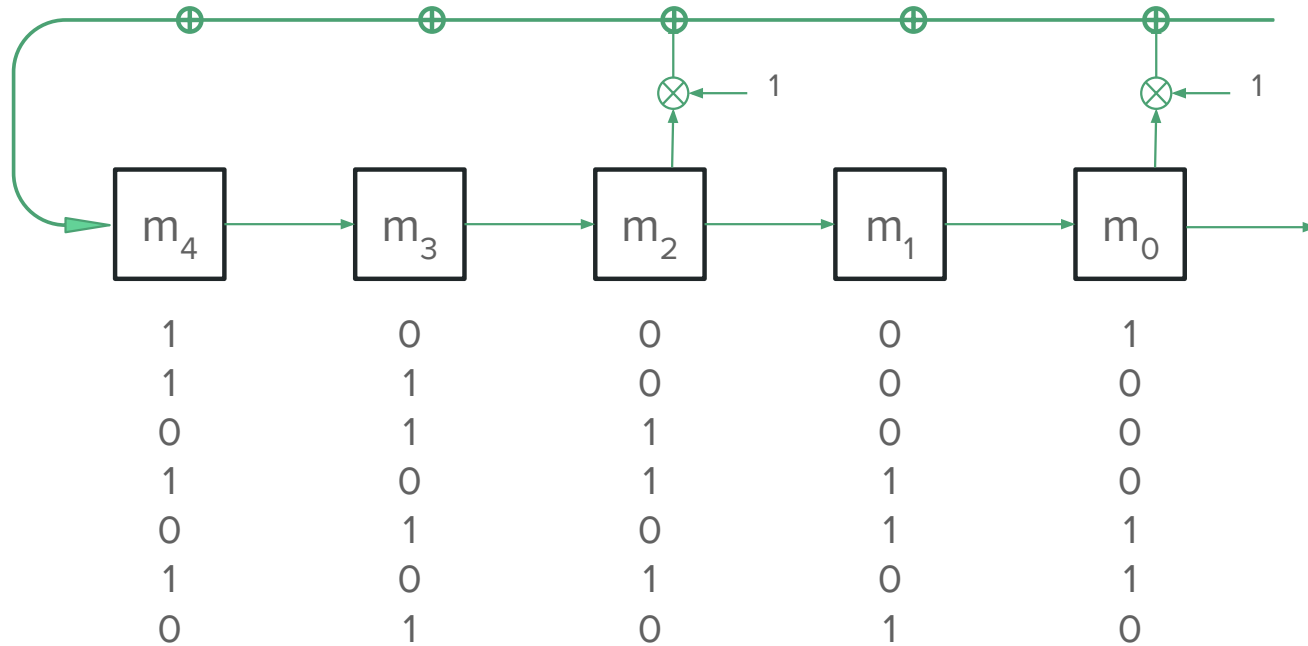
# LFSR - Linear



$$\begin{cases} s_5 = s_4 \otimes p_4 \oplus s_3 \otimes p_3 \oplus s_2 \otimes p_2 \oplus s_1 \otimes p_1 \oplus s_0 \otimes p_0 \\ s_6 = s_5 \otimes p_4 \oplus s_4 \otimes p_3 \oplus s_3 \otimes p_2 \oplus s_2 \otimes p_1 \oplus s_1 \otimes p_0 \\ s_7 = s_6 \otimes p_4 \oplus s_5 \otimes p_3 \oplus s_4 \otimes p_2 \oplus s_3 \otimes p_1 \oplus s_2 \otimes p_0 \end{cases}$$



# LFSR - Example



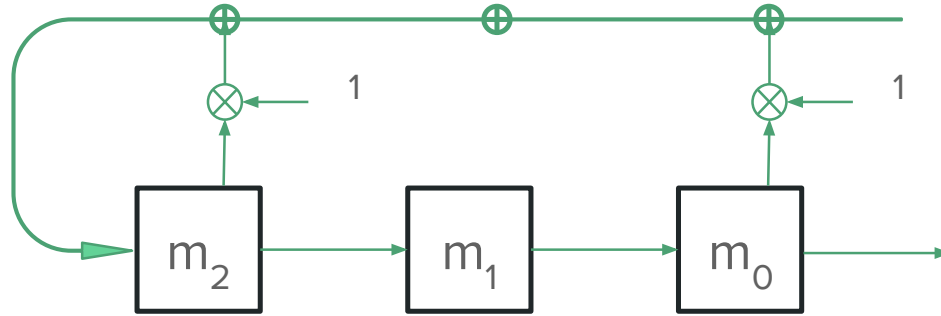
# LFSR - Math

The various configuration of the taps can be summarized with an equation in  $x$  where the greatest degree is equals to the number of registers and each other degree (from  $m-1$  to  $0$ ) represent each tap

$$p = x^m + \sum_{j=m-1}^0 (x^j * p_j)$$

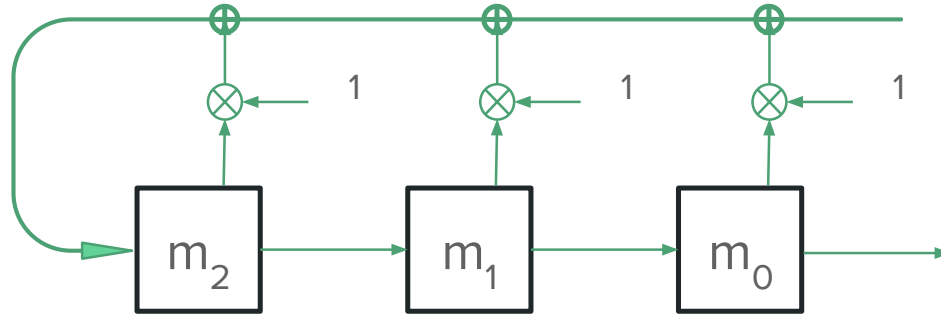
The polynomial  **$p$**  determines the length of the generated sequence. Only **primitive polynomials** produce maximum-length LFSRs, meaning they have no factors.

# LFSR - Math



$$p = x^3 + x^2 + 1$$

# LFSR - Math



$$p = x^3 + x^2 + x + 1$$

Indeed this polynomial is not primitive, one can check that  $p = (x + 1)^3$

## LFSR - Math

As a consequence, the output bits can be described by the following equation:

$$s_{m+i} = \sum_{j=0}^{m-1} s_{i+j} \otimes p_j \bmod 2 \quad i = 0, 1, \dots$$

# LFSR - How to break


As in the previous case, we would like to discover the internal configuration, given only the outputs.

The configuration is the state polynomial  $p$

# LFSR - How to break

Let's consider the case with 3-bit state

$$s_3 = s_2 \otimes p_2 + s_1 \otimes p_1 + s_0 \otimes p_0$$



3 unknowns  
1 equation

# LFSR - How to break

To obtain a determinate system, we need  $m$  equations


$$\begin{cases} s_3 = s_2 \otimes p_2 + s_1 \otimes p_1 + s_0 \otimes p_0 \\ s_4 = s_3 \otimes p_2 + s_2 \otimes p_1 + s_1 \otimes p_0 \\ s_5 = s_4 \otimes p_2 + s_3 \otimes p_1 + s_2 \otimes p_0 \end{cases}$$

As a general rule, to break a LFSR with  $m$ -bits state , we need  **$2m$**  output bits



# Sage-math

Tool to use math functions written in python



```
1  F.<x> = GF(2)[  
2  f = x^3 + x^2 + x + 1  
3  print(f.is_primitive())  
4  print(f.factor())
```

# Useful links

- <https://www.omnicalculator.com/math/linear-feedback-shift-register>
- <https://sagecell.sagemath.org/>
- [Primitive Polynomial](#)

**MT19937**

---

# Mersenne Twister 19937

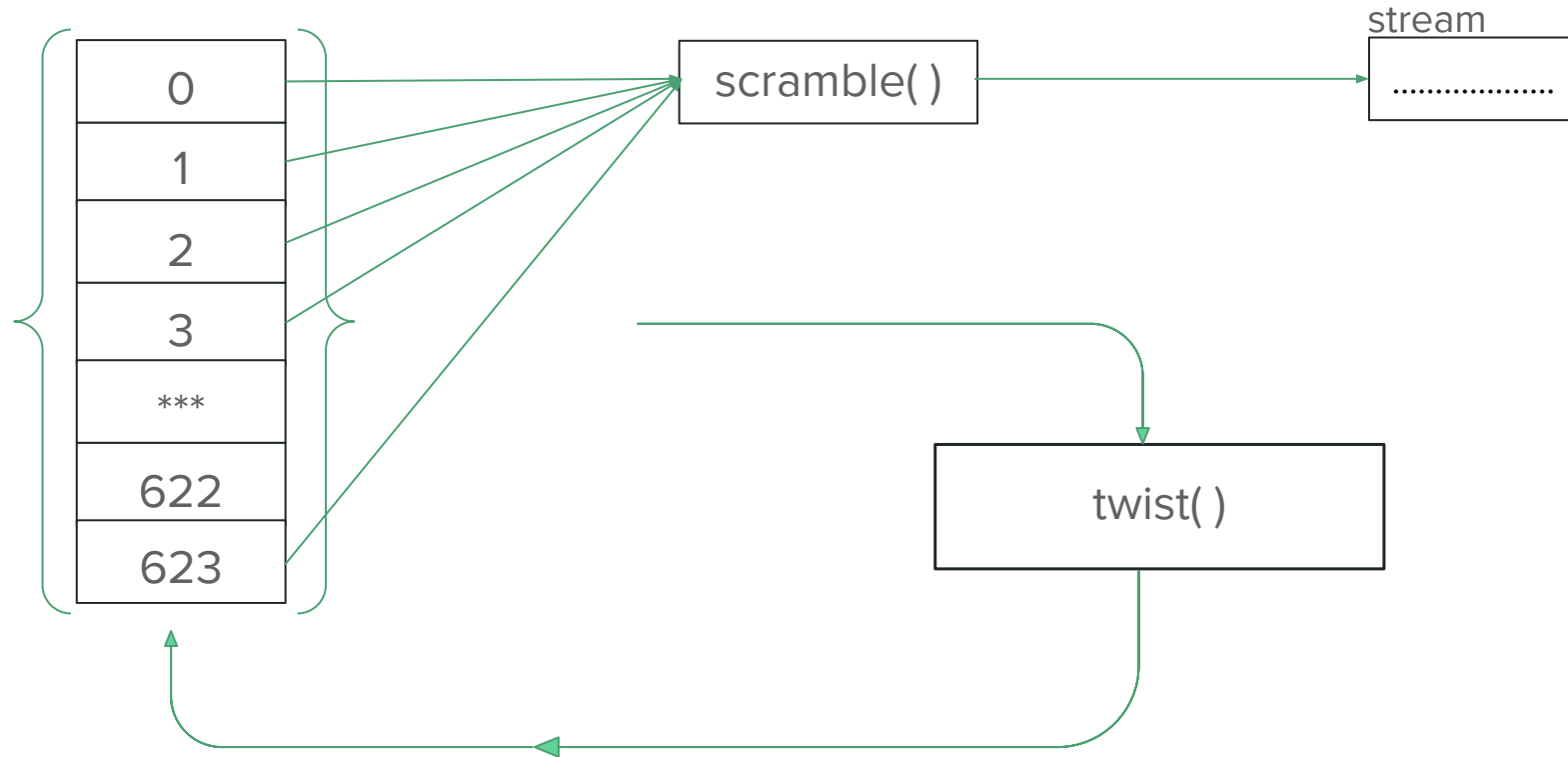
Yet another linear PRNG, developed in 1997 da Makoto Matsumoto e Takuji Nishimura specifically to address most of the flaws found in earlier PRNGs.

The name derives from its period that has been proven to be of  $2^{19937} - 1$ , a Mersenne prime.

The algorithm uses internally a state composed by **624** numbers later combined with various parameters to generate the stream.

Most programs includes this algorithm as option into their random library; in python this is the default behaviour of the package.

# Mersenne Twister 19937



# Mersenne Twister 19937 - scramble( ) function



```
1  y = state[index]
2  y = y ^ ((y >> u) & d)
3  y = y ^ ((y << s) & b)
4  y = y ^ ((y << t) & c)
5  y = y ^ (y >> 1)
6
7  index += 1
8  return y & 0xffffffff
```

```
(u, d) = (11, 0xFFFFFFFF)
(s, b) = (7, 0x9D2C5680)
(t, c) = (15, 0xEFC60000)
l = 18
```

# Mersenne Twister 19937 - twist( ) function



```
1  for i in range(0, n):
2      x = (state[i] & upper_mask) + (state[(i+1) % n] & lower_mask)
3      xA = x >> 1
4      if (x % 2) != 0:
5          xA = xA ^ a
6      state[i] = state[(i + m) % n] ^ xA
```

# Mersenne Twister 19937



Key insights:

- the scramble and twist functions are linear
- ... and easily reversible! (with some effort)
- ... but we need the full 624-int state to do so



# MT19937 - Random

Python's Random module uses internally the MT19937 by exposing some functions:

- seed()
- randint()
- randrange()
- getrandbits() 
- randbytes() 
- shuffle()
- choice()
- ...

# Useful links

- <https://github.com/kmyk/merzenne-twister-predictor>
- <https://github.com/anneouyang/MT19937>
- [https://github.com/icemonster/symbolic\\_mersenne\\_cracker](https://github.com/icemonster/symbolic_mersenne_cracker)

# Challenge time

# Extra

---

# Z3-solver

# What is Z3?

<https://github.com/Z3Prover/z3>

Theorem prover

Efficient SMT solver

***MAGIC!***

Symbolic logic engine

# A “model”?

Symbolic variables

$(x)$   $(x)$   
 $(x)$   $(x)$

Operations and constraints



$(x) \geq 42$

# SMT solver

Problem of determining whether a mathematical formula is *satisfiable*, i.e. there **exists solutions given the relationships between the variables and the defined constraints**

Example

$x \ \& \ y \ \& \ (x \wedge z)$



<b>x</b>	<b>y</b>	<b>z</b>	<b><math>x \&amp; y \&amp; (x \wedge z)</math></b>
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

# We do security, not math. So?

- Formal testing of functional requirements (input -> output)
- Automatic static analysis looking for vulnerabilities
- Symbolic execution
- Automate finding of input to reach a specific point of the program (i.e. the “win” function)

Real example:     **angr**  
(true, real, magic)

<https://angr.io/>

[SMT Solvers for Software Security](#)

# Z3 with Python

<https://ericpony.github.io/z3py-tutorial/>

```
pip install z3-solver
```

Recall math problems at high school?

1. Define variables  
and create a “solver”
2. Add constraints from your problem
3. Solve!

```
from z3 import *

# Define vars and solver
x = BitVec('x', 1)
y = BitVec('y', 1)
z = BitVec('z', 1)
s = Solver()

# Add constraints
s.add((x & y & (x ^ z) == 1))

# Solve!
if s.check() == sat:
    m = s.model()
    print(m)
```

# Play with z3

Try to win again some challenges with z3, in particular

- LCG-2 and LCG-3
- Mersenne Twister