Stream Ciphers

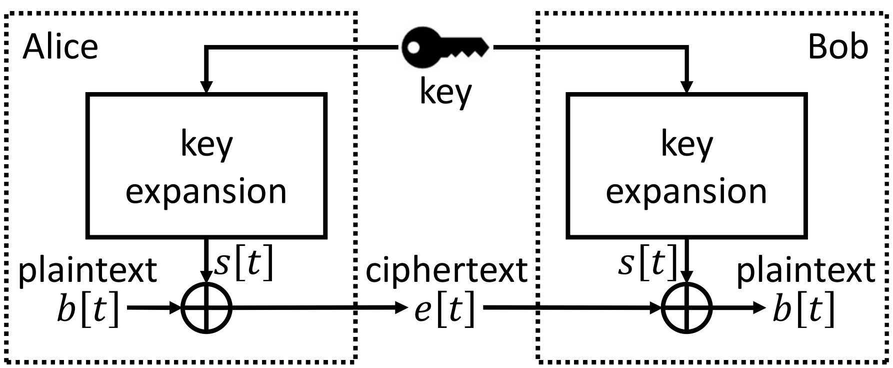
Elements of Applied Data Security M

Alex Marchioni – alex.marchioni@unibo.it

Lorenzo Capelli – l.capelli@unibo.it

Stream Cipher

A stream cipher is a **symmetric key** cipher where the plaintext is encrypted (and ciphertext is decrypted) one digit at a time. A digit usually is either a bit or a byte.



Encryption (decryption) is achieved by xoring the plaintext (ciphertext) with a stream of pseudorandom digits obtained as an expansion of the key.

Tasks

1. Bits class implementation
2. LFSR
3. Berlekamp-Massey Algorithm
4. LFSR-based generator
5. Bonus Task: KPA to LFSR

Task 1: Bits

Bits

Create a class that implements a mutable sequence of bits.

The class must be implemented in a Python module `bits.py`. This class will facilitate the rest of the assignment.

Inputs:

- **bits**: the input is provided either as:
  - Iterable of Booleans: `Bits([True, False, True, True]) -> 1011`
  - Integer: `Bits(0x6a) -> 1101010, Bits(0x6a, length=8) -> 01101010`
  - bytes string: `Bits(b'a') -> 01100001`

Attributes:

- **`bits`**: list of booleans

Bits

Methods (fill free to add any method useful for the assignment):

- **`__len__`**: returns the length of the bit sequence
- **`__str__`**, **`__repr__`**: returns a string of '0' and '1' to be printed/displayed
- **`__getitem__`**, **`__setitem__`**: get/set the value at a given index
- **`__xor__`**, **`__and__`**: computes bit-wise xor/and between two Bits objects
- **`__add__`**: concatenates two Bits objects
- **`__mul__`**: replicates the Bits object by a scalar value (as for lists/tuples)
- **`to_bytes`**: convert the bit sequence into a bytes string/integer
- **`append`**, **`pop`**: append/pop a single bit to the sequence (as for lists)
- **`parity_bit`**: computes parity bit of the bit sequence

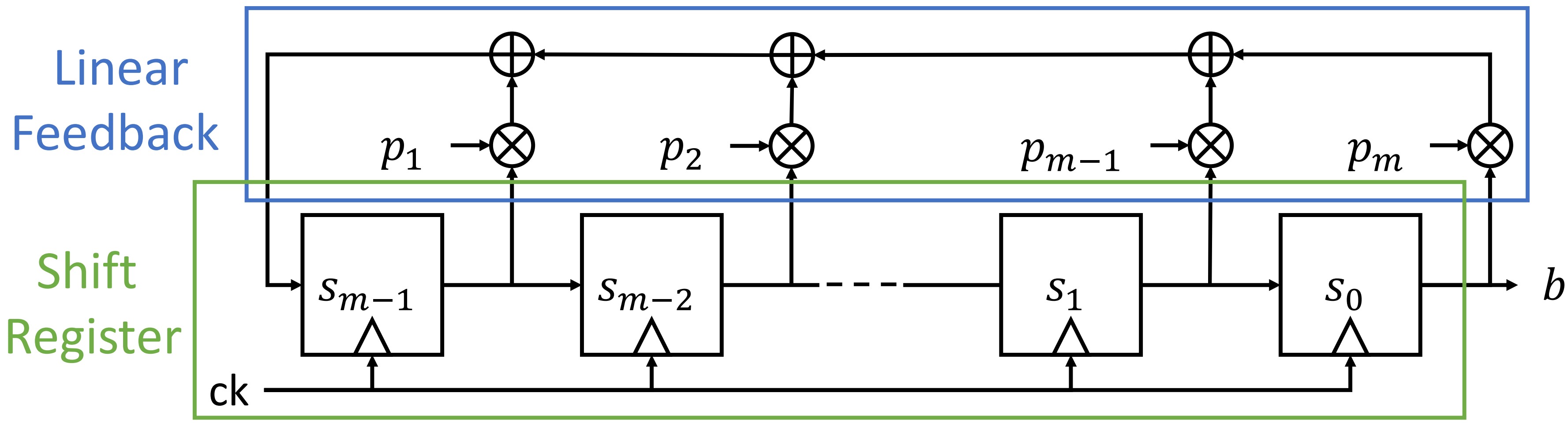
Bits Template

```python
class Bits:
  ''' class docstring '''

  def __init__(self, value, length=None):
    ...
    self.bits = ...

  def __getitem__(self, index):
    ...
    return bit

  def __setitem__(self, index, value):
    ...
    return bit

 def parity_bit(self):
    ...
    return bit

  def __len__(self):
    ...
    return self

  def __str__(self):
    ...
    return string

  def __repr__(self):
    ...
    return string

 def append(self, bit):
    ...
    return bit

 def pop(self, index=-1):
    ...
    return bit

  def __xor__(self, other):
    ...
    return result

  def __and__(self, other):
    ...
    return result
  def __add__(self, other):
    ...
    return result

  def __mul__(self, scalar):
    ...
    return result
```

Task 2: LFSR

LFSR

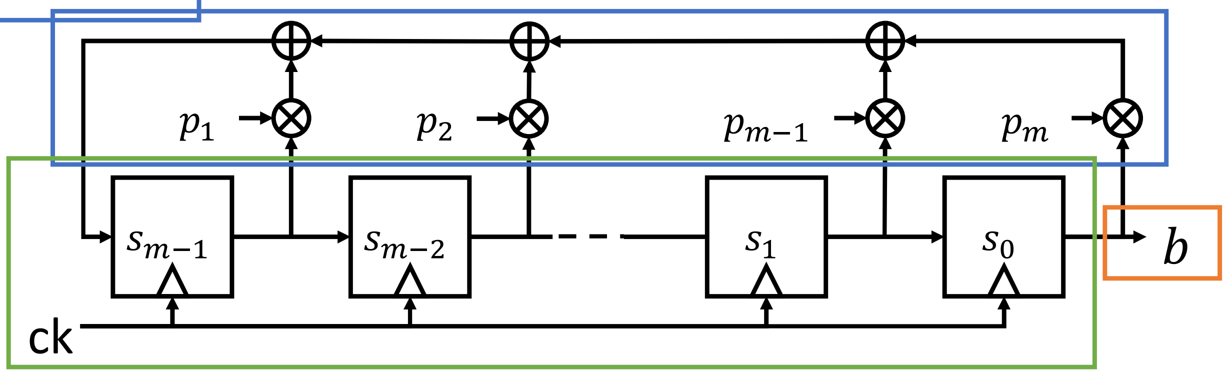
In an LFSR, the output from a standard shift register is fed back into its input causing an endless cycle. The feedback bit is the result of a linear combination of the shift register content and the polynomial coefficients.



LFSR

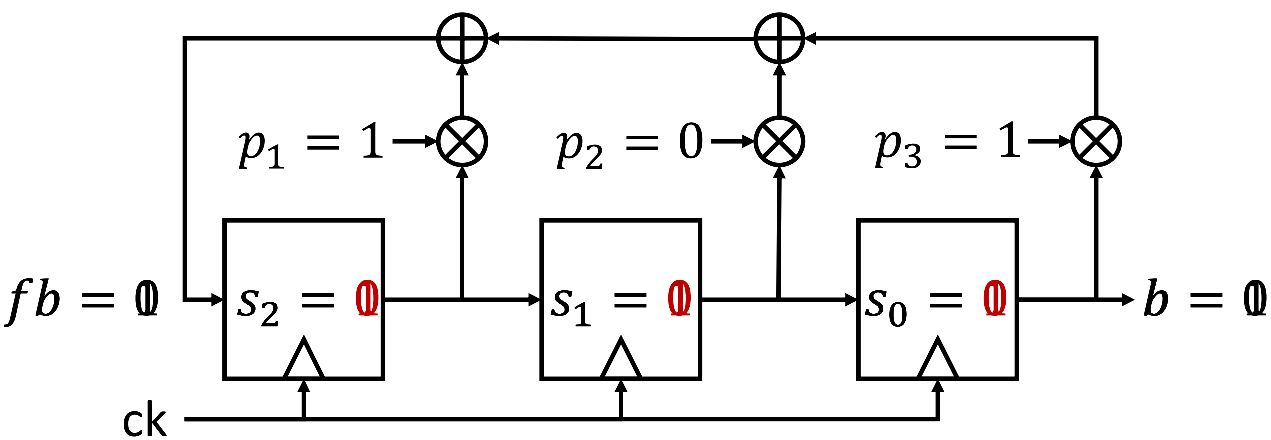
From the block scheme:

$$
\begin{cases}
\boxed{s_j[t] = s_{j+1}[t-1], \qquad j = 0,1,\dots,m-2} \\
\boxed{s_{m-1}[t] = \bigoplus_{j=0}^{m-1} p_{m-j} \otimes s_j[t-1]} \\
\boxed{b[t] = s_0[t]}
\end{cases}
$$



LFSR example

- length = 3
- polynomial = $x^3 + x + 1$ ($p = 0b1011$)
- initial state $0b111$



s	b	fb
111 (7)	1	0
011 (3)	1	1
101 (5)	1	0
010 (2)	0	0
001 (1)	1	1
100 (4)	0	1
110 (6)	0	1
111 (7)	1	0

LFSR Iterable

Create a class that implements an LFSR as an iterable.

The class must be implemented in a Python module `lfsr.py`.

Inputs:

- **Feedback Polynomial**:
  set of integers representing the degrees of the non-zero coefficients. In general, the list is unordered.
  Example: $\{4, 1, 12, 0, 6\}$ represents $P(x) = x^{12} + x^6 + x^4 + x^1 + 1$
- **LFSR state** (optional, default all bits to 1)
  the LFSR initial state provided as either a bytes string, integer, or an iterable of booleans.

LFSR Iterable

Attributes (fill free to add any attribute useful for the assignment):

- **poly**: set of the degrees of the non-zero coefficients of the polynomial (set of int)
- **length**: polynomial degree and length of the shift register (int)
- **state**: LFSR state (Bits)
- **output**: output bit (bool)
- **feedback**: last feedback bit (bool)

LFSR Iterable

Methods (fill free to add any method useful for the assignment):

- **`__init__`**: class constructor
- **`__iter__`**: necessary to be an iterable
- **`__next__`**: update LFSR state and returns the output bit
- **`cycle`**: returns a Bits object representing the full LFSR cycle starting from the current state, or the state given as an argument (pay attention to the case of non-primitive polynomials)
- **`run_steps`**: starting from the current state (or the state given as an optional argument) execute N (optional argument, default N=1) LFSR iterations and returns the corresponding output bits as a Bits object.
- **`__str__`**: return a string describing the LFSR class instance.

LFSR Iterable

```python
class LFSR:

  def __init__(self, poly, state=None):
    ...
    self.poly = ...
    self.length = ...
    self.state = ...
    self.output = ...
    self.feedback = ...

  def __iter__(self):
    return self

  def __str__(self):
    ...
    return string
```

```python
  def __next__(self):
    ...
    return self.output

  def run_steps(self, N=1, state=None):
    ...
    return bits

  def cycle(self, state=None):
    ''' cycle docstring '''
    ...
    return bits
```

Task 3: Berlekamp-Massey Algorithm

Berlekamp-Massey Algorithm

Find the shortest LFSR for a given binary sequence.

- **Inputs**: sequence of bit b of length N
- **Outputs**: feedback polynomial $P(x)$.

```
def berlekamp_massey(bits):
  ''' function docstring '''
  # algorithm implementation
  # bits as a Bits object
  return poly
```

Input $b = [b_0, b_1, \dots, b_N]$
$P(x) \leftarrow 1, m \leftarrow 0$
$Q(x) \leftarrow 1, r \leftarrow 1$
For $\tau = 0,1, \dots, N-1$

$$d \leftarrow \bigoplus_{j=0}^{m} p_j \otimes b[\tau - j]$$

    If $d = 1$ **then**
        If $2m \leq \tau$ **then**
            $R(x) \leftarrow P(x)$
            $P(x) \leftarrow P(x) + Q(x)x^r$
            $Q(x) \leftarrow R(x)$
            $m \leftarrow \tau + 1 - m$
            $r \leftarrow 0$
        else
            $P(x) \leftarrow P(x) + Q(x)x^r$
        endif
    endif
    $r \leftarrow r + 1$
endfor
Output $P(x)$

Berlekamp-Massey Algorithm

τ	b_τ	d		$P(x)$	m	$Q(x)$	r
-	-	-		1	0	1	1
0	1	1	A	$1+x$	1	1	1
1	0	1	B	1	1	1	2
2	1	1	A	$1+x^2$	2	1	1
3	0	0		$1+x^2$	2	1	2
4	0	1	A	1	3	$1+x^2$	1
5	1	1	B	$1+x+x^3$	3	$1+x^2$	2
6	1	0		$1+x+x^3$	3	$1+x^2$	3
7	1	0		$1+x+x^3$	3	$1+x^2$	4

Input $b = [b_0, b_1, \dots, b_N]$
$P(x) \leftarrow 1, m \leftarrow 0$
$Q(x) \leftarrow 1, r \leftarrow 1$
For $\tau = 0,1,\dots,N-1$
$\quad d \leftarrow \bigoplus_{j=0}^{m} p_j \otimes b[\tau - j]$
$\quad$ **If** $d = 1$ **then**
$\quad\quad$ **If** $2m \leq \tau$ **then**

A
$\quad\quad\quad R(x) \leftarrow P(x)$
$\quad\quad\quad P(x) \leftarrow P(x) + Q(x)x^r$
$\quad\quad\quad Q(x) \leftarrow R(x)$
$\quad\quad\quad m \leftarrow \tau + 1 - m$
$\quad\quad\quad r \leftarrow 0$

$\quad\quad$ **else**

B
$\quad\quad\quad P(x) \leftarrow P(x) + Q(x)x^r$

$\quad\quad$ **endif**
$\quad$ **endif**
$\quad r \leftarrow r + 1$
endfor
Output $P(x)$

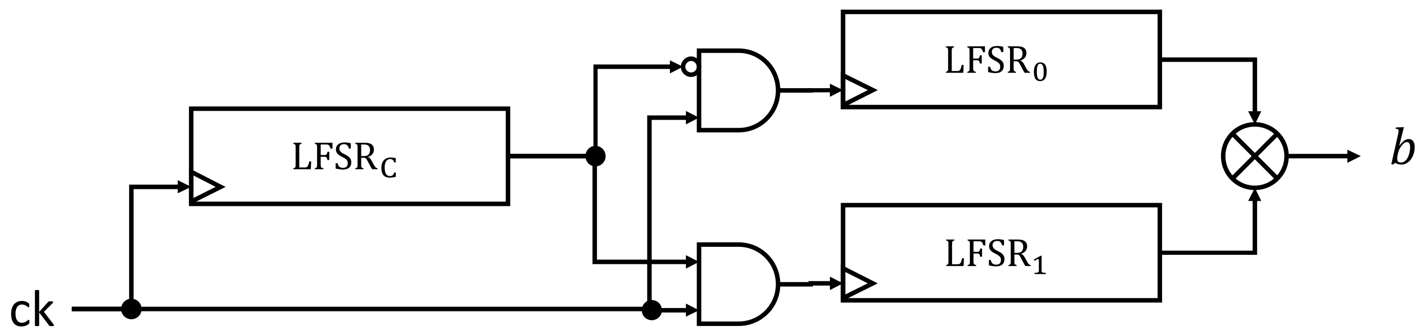
Berlekamp-Massey Algorithm Task

- Implement the Berlekamp-Massey Algorithm as a function in `lfsr.py`.
- Apply the Berlekamp-Massey Algorithm to the bit sequence stored in the file `binary_sequence.bin` to compute:
  - The polynomial of the shortest LFSR that can produce that sequence
  - The linear complexity of the bit sequence

Task 4:
LFSR-based generator

Alternating-Step Generator

Three LFSR of which $\mathrm{LFSR_C}$ decides which between $\mathrm{LFSR_0}$ and $\mathrm{LFSR_1}$ is clocked. The output is the XOR of $\mathrm{LFSR_0}$ and $\mathrm{LFSR_1}$current outputs.



Alternating-Step Generator Class

Create a class that implements an Alternating-Step Generator as an iterable.

The class must be implemented in a Python module `bitgenerator.py`.

- **Inputs**
  - Seed `seed` which must be used to initialize the polynomial states as follows: first bits to initialize $\mathrm{LFSR_C}$, then $\mathrm{LFSR_0}$, last $\mathrm{LFSR_1}$. It is an optional argument; default is all bits of the LFSR states at 1.
  - Polynomials `polyC`, `poly0` and `poly1`. They are optional arguments, defaults are
    - $P_C(x) = x^5 + x^2 + 1$
    - $P_0(x) = x^3 + x + 1$
    - $P_1(x) = x^4 + x + 1$
- **Attributes**
  - `lfsrC`, `lfsr0`, `lfsr1`, the LFSR class instances for $\mathrm{LFSR_C}$, $\mathrm{LFSR_0}$, and $\mathrm{LFSR_1}$.
  - `output`: boolean storing the last produced output bit

Example of an Alternating-Step Generator

- Polynomials: $P_C(x) = x^5 + x^2 + 1$, $P_0(x) = x^3 + x + 1$, $P_1(x) = x^4 + x + 1$
- States: all bits set to 1

t	LFSR_C	b_C	LFSR_0	b_0	LFSR_1	b_1	b
0	11111 (1f)	1	111 (7)	1	1111 (0f)	1	
1	01111 (0f)	1	111 (7)	1	0111 (07)	1	0
2	00111 (07)	1	111 (7)	1	1011 (0b)	1	0
3	10011 (13)	1	111 (7)	1	0101 (05)	1	0
4	11001 (19)	1	111 (7)	1	1010 (0a)	0	1
5	01100 (0c)	0	011 (3)	1	1010 (0a)	0	1
6	10110 (16)	0	101 (5)	1	1010 (0a)	0	1
7	01011 (0b)	1	101 (5)	1	1101 (0d)	1	0
8	00101 (05)	1	101 (5)	1	0110 (06)	0	1
9	10010 (12)	0	010 (2)	0	0110 (06)	0	0
10	01001 (09)	1	010 (2)	0	0011 (03)	1	1
11	00100 (04)	0	001 (1)	1	0011 (03)	1	0
12	00010 (02)	0	100 (4)	0	0011 (03)	1	1
13	00001 (01)	1	100 (4)	0	1001 (09)	1	1
14	10000 (10)	0	110 (6)	0	1001 (09)	1	1
15	01000 (08)	0	111 (7)	1	1001 (09)	1	0
16	10100 (14)	0	011 (3)	1	1001 (09)	1	0
17	01010 (0a)	0	101 (5)	1	1001 (09)	1	0
18	10101 (15)	1	101 (5)	1	0100 (04)	0	1
19	11010 (1a)	0	010 (2)	0	0100 (04)	0	0
20	11101 (1d)	1	010 (2)	0	0010 (02)	0	0
21	01110 (0e)	0	001 (1)	1	0010 (02)	0	1
22	10111 (17)	1	001 (1)	1	0001 (01)	1	0
23	11011 (1b)	1	001 (1)	1	1000 (08)	0	1
24	01101 (0d)	1	001 (1)	1	1100 (0c)	0	1
25	00110 (06)	0	100 (4)	0	1100 (0c)	0	0

Alternating-Step Generator Class Template

```
class AlternatingStep:

    def __init__(self, seed=None, polyC=None, poly0=None, poly1=None):
        self.lfsrC = LFSR(...)
        self.lfsr0 = LFSR(...)
        self.lfsr1 = LFSR(...)
        self.output = ...

    def __iter__(self):
        return self

    def __next__(self):
        ...
        return self.output
```

Bonus Task: KPA to LFSR

Bonus Task

- Perform a Known Plaintext Attack (KPA) to a Stream Cipher in which the key stream is generated with an LFSR with an unknown structure.
- Decrypt the bit sequence in `ciphertext.bin` having the initial portion of the original text in `known-plaintext.txt`.
  Use utf-8 encoded if needed.
- Return:
  - the whole original plaintext,
  - the `poly` that defines the LFSR.

`known-plaintext.txt`

```
The Legacy of the Hidden Key

In a quiet corner of the university
library, where dust motes danced in the
slanted afternoon light, Jamie discovered
an old, leather-bound journal tucked
behind a row of forgotten books. Its
cover bore no title, only an intricate
emblem reminiscent of interlocking keys.
Curiosity sparked, Jamie carefully opened
the journal and found pages filled with
cryptic symbols and strange, archaic
notations—a mysterious cipher, perhaps
long lost to time.
```

Deadline

Tuesday, April 29th at 12PM (noon)