Stream Ciphers

Elements of Applied Data Security M

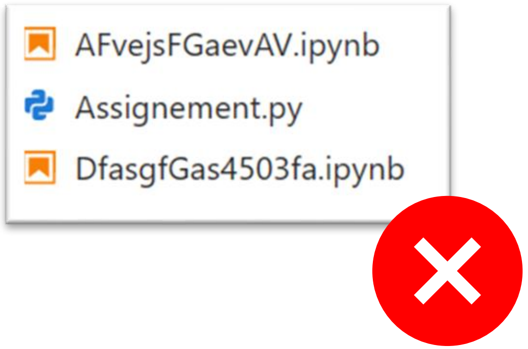
Alex Marchioni– alex.marchioni@unibo.it

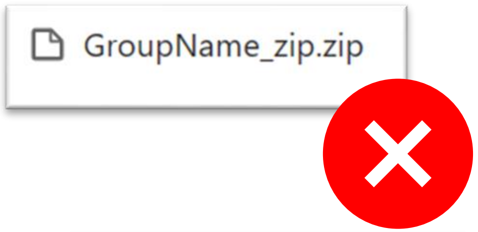
Filippo Martinini– filippo.martinini@unibo.it

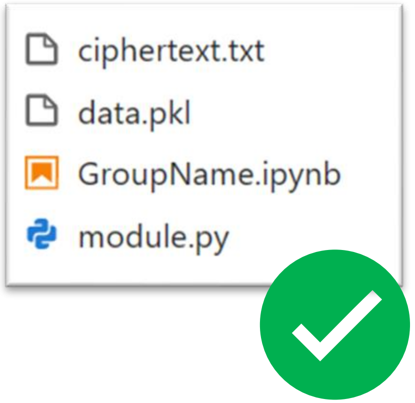
Andriy Enttsel– andriy.enttsel@unibo.it

About Assignment 1

- assignments must be in **notebook** format
  - Report must be in a notebook and not in “.py” file. “.py” files are allowed as modules, only to store functions and they must be imported in the main notebook.
  - Only one notebook is allowed (other notebooks will be ignored).
  - The notebook name must be the name of the group.
- Do zip your files but upload them separately.
- All files required to run your notebook must be uploaded.
  - For example, “ciphertext.txt” must be uploaded together with the notebook.
  - If the notebook does not run for a missing file, it will be considered as an error.







About Assignment 1

- Remove template indications:
  - Write the report as you were going to submit it to your boss at the company you are working for.
- Report must be written in markdown cells:
  - Formatting (bold, italic, formulas in latex) and inclusion of figures are appreciated.
  - However, formatting must be used wisely.
  - Respect the structure of the template. You can add subsections but be sure to be coherent with the notebook organization.

✓ **Title**

✓ **Subsection**

✗ `Report written in Raw Cells is not nice to read`

✓ Instead write in Markdown Cells

✓ To highlight part of the text use **bold**, or *italic*.

✗ **DO not use title/subtitle formatting to write text!!**

✗ Description (Max 150 words) Your description

✓ Your description

About Assignment 1

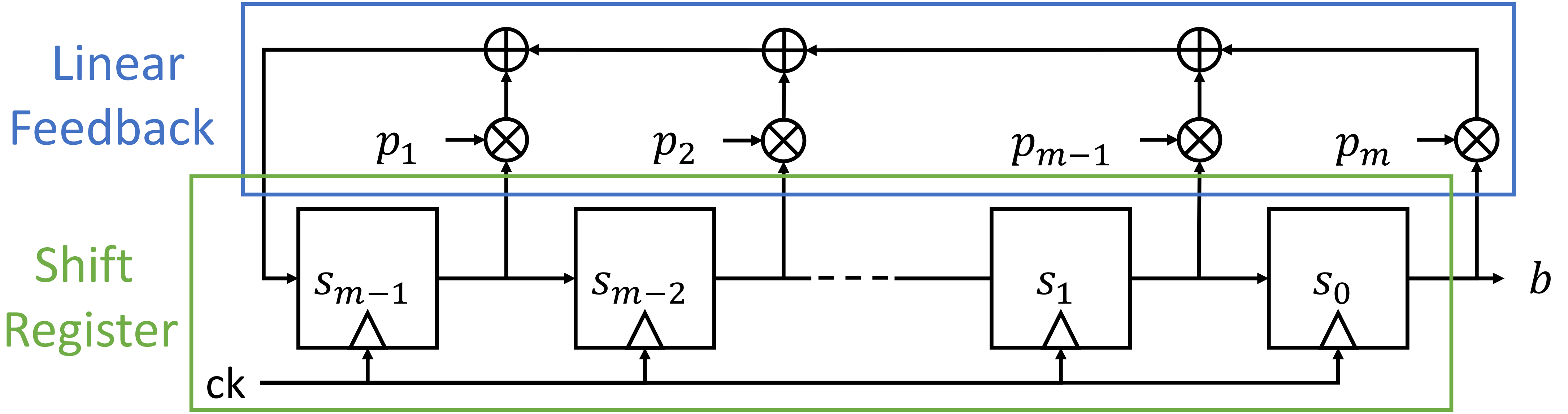
- We encourage collaboration but not plagiarism
  - Collaboration is encouraged, we found many similarities in the notebooks submitted by different groups: that is completely fine. In our opinion, it means you are actively discussing the project with each other.
  - However, each group must produce an original notebook. Plagiarism will be marked as a 0.

Task 1: LFSR

- Implement an LFSR.

LFSR

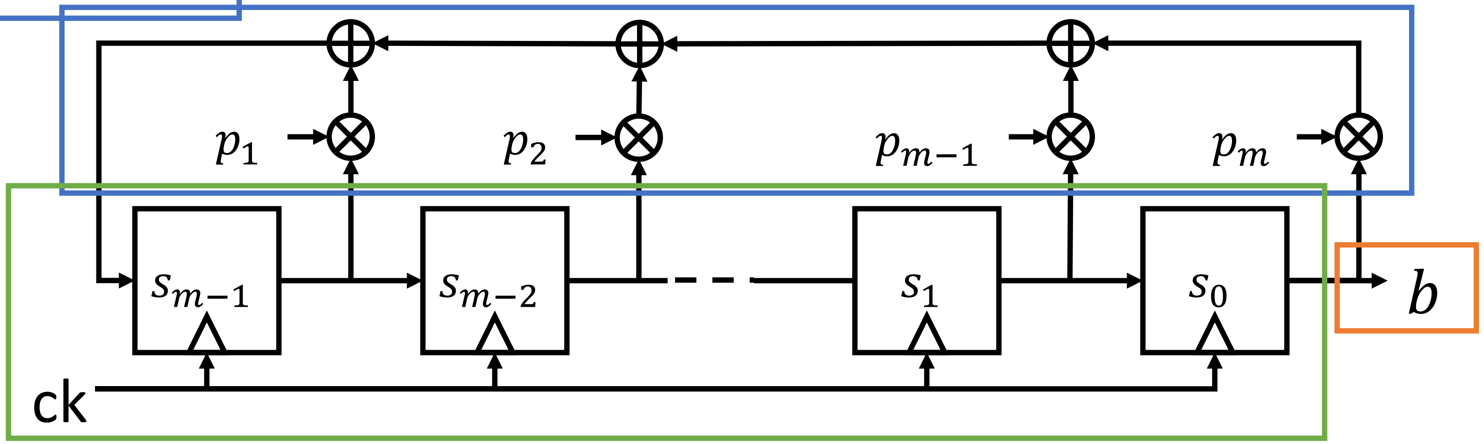
In an LFSR, the output from a standard shift register is fed back into its input causing an endless cycle. The feedback bit is the result of a linear combination of the shift register content and the polynomial coefficients.



LFSR

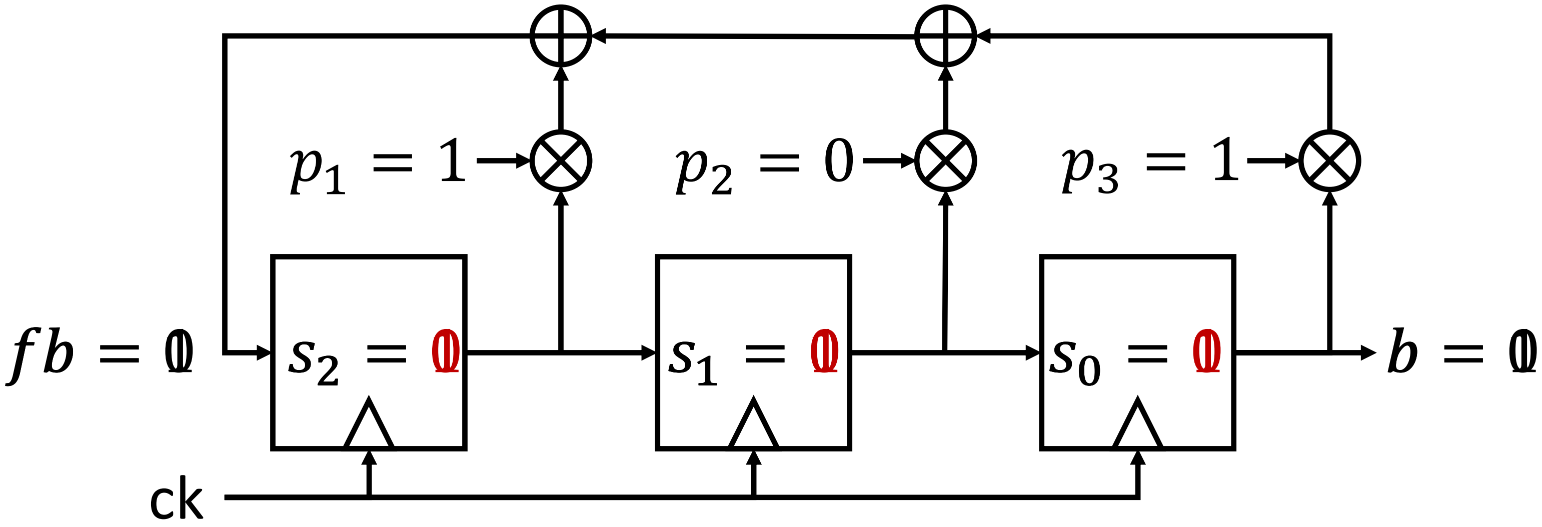
From the block scheme:

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



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

Inputs:

- **Feedback Polynomial**:
  list of integers representing the degrees of the non-zero coefficients.
  Example: $[12, 6, 4, 1, 0]$ represents $x^{12} + x^6 + x^4 + x^1 + 1$
- **LFSR state** (optional, default all bits to 1)
  Integer or bitstream representing the LFSR initial state
  Example: 0xA65 for [1010 0110 0101]

LFSR Iterable

Attributes:

- **poly**: list of the polynomial coefficients (list of int)
- **length**: polynomial degree and length of the shift register (int)
- **state**: LFSR state (int or list of bits)
- **output**: output bit (bool)
- **feedback**: last feedback bit (bool)

LFSR Iterable

Methods:

- **__init__**: class constructor;
- **__iter__**: necessary to be an iterable;
- **__next__**: update LFSR state and returns output bit;
- **cycle**: returns a list of bool representing the full LFSR cycle ;
- **run_steps**: execute N LFSR steps and returns the corresponding output list of bool (N is a input parameter, default N=1);
- **__str__**: return a string describing the LFSR class instance.

LFSR Iterable

```python
class LFSR(object):
  ''' class docstring '''

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

  def __iter__(self):
    return self

  def __next__(self):
    ''' next docstring '''
    ...
    return self.output

  def run_steps(self, N=1):
    ''' run_steps docstring '''
    ...
    return list_of_bool

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

Hints

There are many ways to implement an LFSR in Python.

The first choice to make is how to store the internal state and the polynomial. I suggest two types:

- **list of bool**: it is the most straightforward choice as it directly maps the LFSR block scheme, but bit-wise logical operation may not be as easy.
- **integer**: bit-wise logical operation, as well as bit-shift, are easy to perform on integers, while XOR of multiple bits or reversing the bit order are less straightforward.

Useful functions

- **XOR**: In Python bit-wise xor between two integers is implemented with the `^` mark. It is also implemented as function (`xor`) in the built-in module [`operator`](). Example: `xor(5,4) -> 5^4 -> 0b101^0b100 -> 0b001 -> 1`
- **reduce**: available from the built-in module [`functools`](), apply a function of two arguments cumulatively to the items of an iterable so as to reduce the iterable to a single value.
  Example: `reduce(xor, [True, False, True, False]) -> False`
- **compress**: available from the built-in module [`itertools`](), make an iterator that filters elements from data returning only those that have a corresponding element in selectors that evaluates to True.
  Example: `compress([3, 7, 5], [True, False, True]) -> [3, 5]`

Task 2: Berlekamp-Massey Algorithm

- Implement the Berlekamp-Massey Algorithm.
- Use the Berlekamp-Massey Algorithm to compute the linear complexity of a bit sequence.

Berlekamp-Massey Algorithm

Find the shortest LFSR for a given binary sequence.

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

```python
def berlekamp_massey(b):
  ''' function docstring '''
  # algorithm implementation
  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 $\tau = 0,1,\dots,N-1$
    $d \leftarrow \bigoplus_{j=0}^{m} p_j \otimes b[\tau - j]$
    If $d = 1$ **then**
        If $2m \leq \tau$ **then**
A
            $R(x) \leftarrow P(x)$
            $P(x) \leftarrow P(x) + Q(x)x^r$
            $Q(x) \leftarrow R(x)$
            $m \leftarrow \tau + 1 - m$
            $r \leftarrow 0$
        else
B
            $P(x) \leftarrow P(x) + Q(x)x^r$
        endif
    endif
    $r \leftarrow r + 1$
endfor
Output $P(x)$

Hints

There are many ways to implement BM in Python.

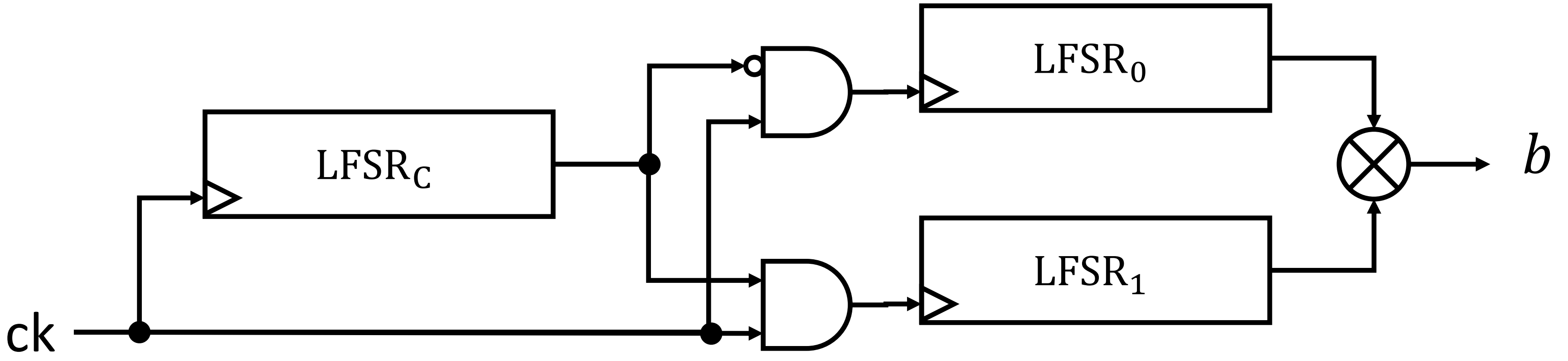
We suggest an implementation based on **integers** since bit-wise logical operation, as well as bit-shift, are easy to perform on integers, while XOR of multiple bits or reversing the bit order are less straightforward.

Task 3: LFSR-based generator

- Implement the Alternating Step Generator.
- Use the Berlekamp-Massey algorithm to compute the linear complexity of the generated sequence.
- Use the Alternating Step Generator as a CPRNG to decrypt a message.

Alternating-Step Generator

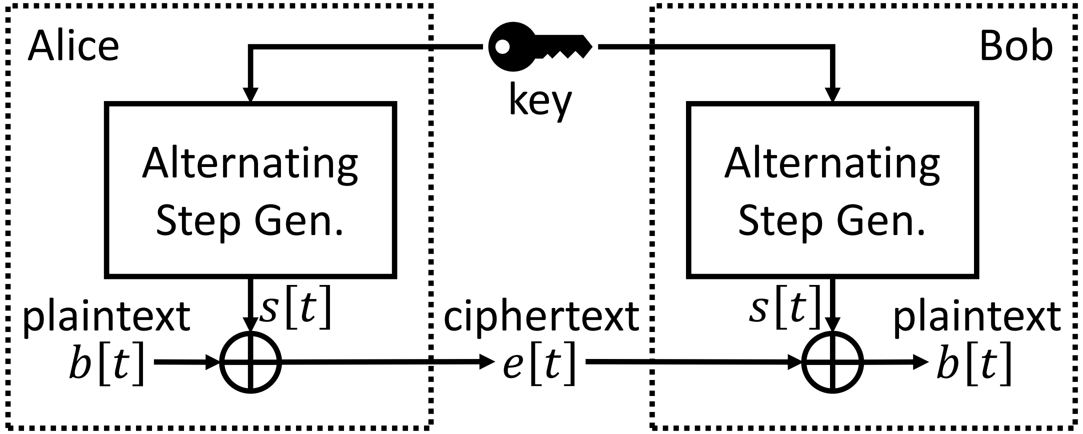
Three LFSR of which $\mathrm{LFSR_C}$ decides which between $\mathrm{LFSR_0}$ and $\mathrm{LFSR_1}$ is clocked. The output is the XOR of $\mathrm{LFSR_0}$ and $\mathrm{LFSR_1}$ current outputs.



$$P_0(x) = x^5 + x^2 + 1$$
$$P_1(x) = x^3 + x + 1$$
$$P_C(x) = x^2 + x + 1$$

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.

Task 4: RC4

Rivest Cipher 4 (RC4)

RC4 is a stream cipher that generates the keystream from a secret internal state which consists of two parts:

- A permutation P of all 256 possible bytes.
- Two 8-bit index-pointers (denoted i and j).

P is initialized with a variable length key by means of the key-scheduling algorithm (KSA).

Then, the keystream is generated using the pseudo-random generation algorithm (PRGA) that updates the indexes i and j, modifies the permutation P and generates a random byte.

Key Scheduling Algorithm (KSA)

The KSA is used to initialize the permutation P starting from a key composed by L bytes. Typical values for L range from 5 to 16.

Input key $= [k_0, k_1, \dots, k_{L-1}]$,
with $k_i \in \{0, 1, \dots, 255\}$
$j \leftarrow 0$
for $i = 0,1, \dots, 255$
$\quad P[i] \leftarrow i$
endfor
for $i = 0,1, \dots, 255$
$\quad j \leftarrow (j + P[i] + \text{key}[i \bmod L]) \bmod 256$
$\quad P[i], P[j] \leftarrow P[j], P[i]$
endfor
$i, j \leftarrow 0, 0$
Output P

P is initialized with an identity permutation ($P[i] = i$).

Then, bytes of P are mixed iteratively in a way that depends on the key.

Pseudo-random generation Algorithm (PRGA)

For each iteration, PRGA modifies the state (represented by the permutation P and the pair of indexes i, j) and outputs a byte.

$$
\begin{aligned}
&\textbf{State } P, i, j \\
&i \leftarrow (i+1) \bmod 256 \\
&j \leftarrow (j + P[i]) \bmod 256 \\
&P[i], P[j] \leftarrow P[j], P[i] \\
&K \leftarrow P[(P[i] + P[j]) \bmod 256] \\
&\textbf{Output } K
\end{aligned}
$$

In each iteration,

- i is incremented,
- j is updated by adding the value $P[i]$,
- $P[i]$ and $P[j]$ are swapped.
- The output byte is element of P ant the location $P[i] + P[j] \pmod{256}$

RC4-drop[n]

RC4 has many known vulnerabilities mainly related to the correlation between the key and the first bytes of the permutation P.

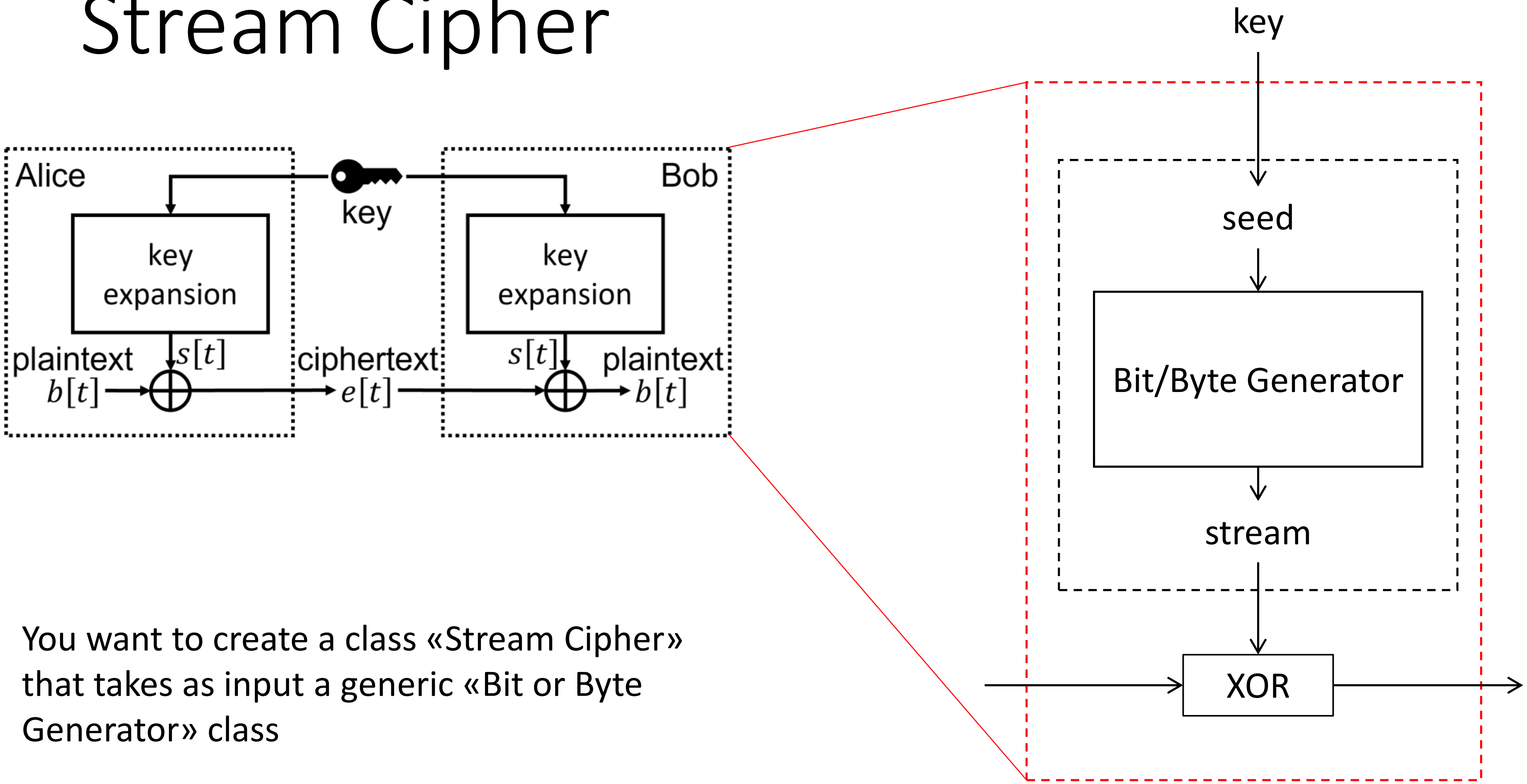
Most of them can be avoided by discarding the first n bytes of the output stream, from where it becomes RC4-drop[n].

Typical values for n are:

- $n = 768$
- $n = 3072$ (more conservative value)

Bonus Task

Stream Cipher



You want to create a class «Stream Cipher» that takes as input a generic «Bit or Byte Generator» class

Stream Cipher

Inputs:

- **key**: integer representing the shared secret key.
- **PRNG**: Iterator implementing a PRNG that produce a pseudorandom bit/byte stream starting from an initial seed.

Methods:

- **encrypt:** encrypts a plaintext (bytes) and returns the corresponding ciphertext (bytes);
- **decrypt:** decrypts a ciphertext (bytes) and returns the corresponding plaintext (bytes);

Stream Cipher

Template:

```python
class StreamCipher():
  ''' class docstring '''

  def __init__(self, key, prng, **kwargs):
    ''' constructor docstring '''
    # do stuff
    self.prng = ...

  def encrypt(self, plaintext):
    # do stuff
    return ciphertext

  def decrypt(self, ciphertext):
    # do stuff
    return plaintext
```

**kwargs

The ****kwargs** are utilized to pass a dictionary of variable-length, keyword arguments to a function. It represents "keyword arguments" and enables the passing of argument dictionaries to a function.

Try this code:

```
def example_function(**kwargs):
    for key, value in kwargs.items():
        print(key, value)

example_function(a=1, b=2, c=3)
```

Stream Cipher Example

```
message = 'hello world!'
key = 0x12345678

# create a StreamCipher instance for Alice and Bob
alice = StreamCipher(key)
bob   = StreamCipher(key)

plaintextA = message.encode('utf-8')   # string to bytes
ciphertext = alice.encrypt(plaintextA) # encryption by Alice
plaintextB = bob.decrypt(ciphertext)   # decryption by Bob
```

DEADLINE

- Tuesday, May 2, 12PM (noon)