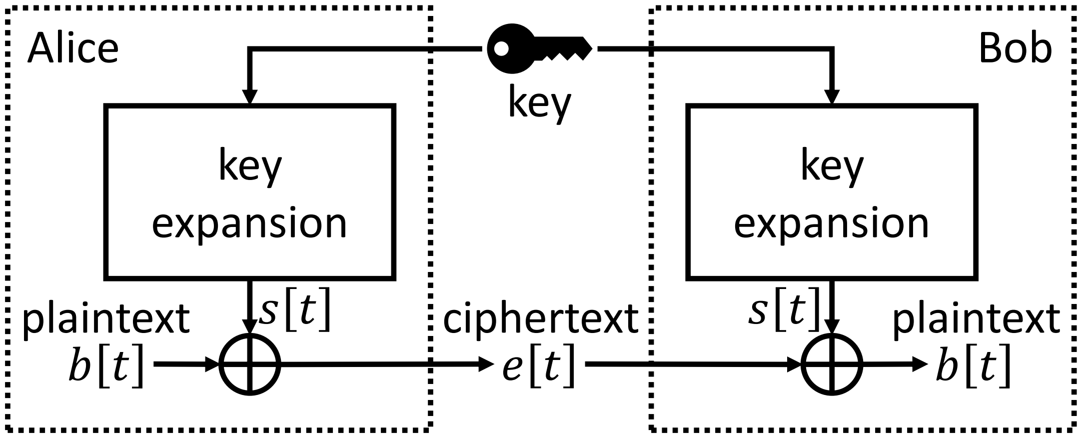
Stream Ciphers

Elements of Applied Data Security

Alex Marchioni

Stream Cipher

Symmetric key cipher where the plaintext is encrypted one digit at a time. Encryption is achieved by xoring the plaintext with a stream of pseudorandom bits obtained as an expansion of the key.



Task

1. Define an object representing a Stream Cipher that, given a PRNG and a key, can encrypt and decrypt a message one bit at a time.
2. Define an iterator that implements a PRNG based on the A5/1 architecture and use it in the Stream Cipher object previously implemented to encrypt/decrypt a message.
3. Add to the StreamCipher class the possibility to encrypt/decrypt a message using bytes as digits instead of bits.
4. Define an iterator that implements a PRNG based on the RC4 architecture and use it in the new Stream Cipher object to encrypt/decrypt a message

Task 1: Stream Cipher

Stream Cipher

Inputs:

- **key**: integer representing the shared secret key.
- **PRNG** (optional, default None): Iterator implementing a PRNG that produce a pseudorandom bit stream starting from an initial seed. If None an LFSR is used as PRNG.

Methods:

- **encrypt:** encrypts a plaintext (str, bytes) and returns the corresponding cyphertext (bytes);
- **decrypt:** decrypts a cypertext (str, bytes) and returns the corresponding plaintext (bytes);

Stream Cipher

Template:

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

  def __init__(self, key, prng=None):
    ''' constructor docstring '''
    # do stuff
    self.prng = ...

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

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

Stream Cipher Example

```
message = 'hello world!'
key = 0x0123456789ABCDEF

# define a function that create an instance of an LFSR
prng = lambda key: LFSR([64, 4, 3, 1, 0], state=key)
# create a StreamCipher instance for both Alice and Bob
alice = StreamCipher(key, prng=prng)
bob   = StreamCipher(key, prng=prng)

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

print(plaintextA) # -> b'hello world!'
print(ciphertext) # -> b'\x87\x02\xc7O\xa2e\xfen\x14\xfb\xafv'
print(plaintextB) # -> b'hello world!'
```

bitstring

bitstring is a pure Python module designed to help creation and analysis of binary data.

It implements the following classes:

- `BitArray`: a container for binary data.
- `BitStream`: adds a bit inserting and reading methods to treat the data as a stream.

Here is a Walkthrough to get started: https://pythonhosted.org/bitstring/walkthrough.html#

Task 2: A5/1

A5/1

- The **stream cipher** used to provide privacy in the **GSM** cellular telephone standard.
- Based on a combination of **three LFSRs with irregular clocking**.

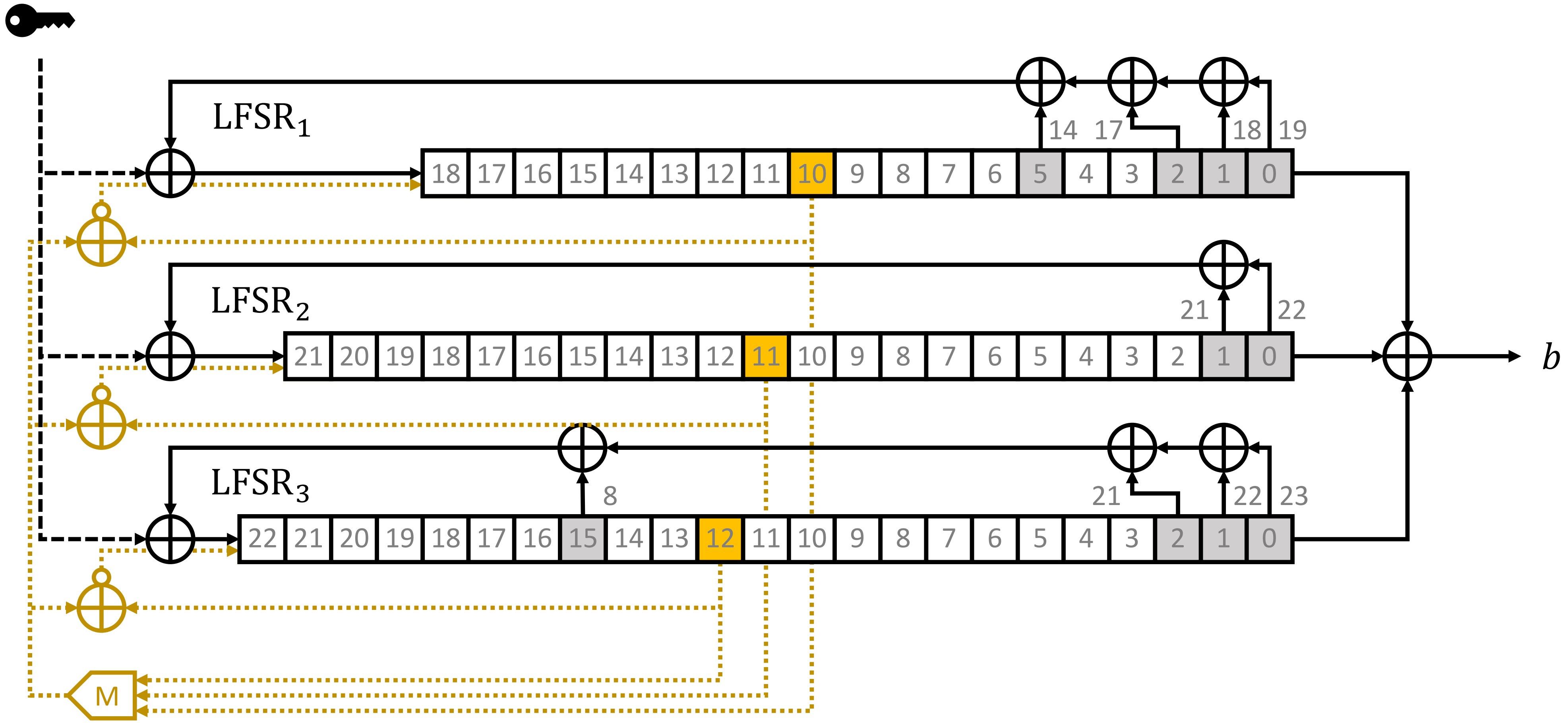
  At each cycle, the clocking bit of all three registers is examined and the majority bit is determined. A register is clocked if the clocking bit agrees with the majority bit.

LFSR	Length	Feedback Polynomial	Clocking bit
1	19	$x^{19}+x^{18}+x^{17}+x^{14}+1$	10
2	22	$x^{22}+x^{21}+1$	11
3	23	$x^{23}+x^{22}+x^{21}+x^{8}+1$	12

A5/1

- A5/1 is initialised using a 64-bit private **key** together with a publicly known 22-bit **frame** number.
  - Initially, the registers are set to zero.
  - The 64-bit secret key is mixed: in cycle i (with $0 \leq i < 64$), the i-th key bit is added to the most significant bit of each register using XOR.
  - Similarly, the 22-bits frame number is added in 22 cycles.
- Then, the cipher is clocked using the normal majority clocking mechanism for 100 cycles, with the output discarded.
- Then, the cipher is ready to produce two 114 bit sequences of output keystream, first 114 for downlink, last 114 for uplink.

A5/1



LFSR$_1$
14 17 18 19
18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
LFSR$_2$
21 22
21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
LFSR$_3$
8
21 22 23
22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
M
b

A5/1

- Define a Iterator that implements the A5/1 stream cipher.
- Given a key and a frame number, encrypt and decrypt a message.

```
message = 'hello world!’
key, frame = 0x0123456789ABCDEF, 0x2F695A

# create a StreamCipher instance for both Alice and Bob
alice = StreamCipher(key, prng=A5_1, frame=frame)
bob = StreamCipher(key, prng=A5_1, frame=frame)

plaintextA = message.encode('utf-8’)   # -> b'Hello world!'
ciphertext = alice.encrypt(plaintextA) # -> b'I\x04\x04\xe8E&\x89Vr_j\x9d'
plaintextB = bob.decrypt(ciphertext)   # -> b'Hello world!'
```

A5/1 Key insertion

iter	key	LFSR1	LFSR2	LFSR3	maj	out
1	1	40000	200000	400000	0	0
2	1	60000	300000	600000	0	0
3	1	70000	380000	700000	0	0
4	1	78000	3c0000	780000	0	0
5	0	3c000	1e0000	3c0000	0	0
6	1	5e000	2f0000	5e0000	0	0
7	1	6f000	378000	6f0000	0	0
8	1	77800	3bc000	778000	0	0
9	1	7bc00	3de000	3bc000	0	0
10	0	3de00	1ef000	5de000	0	0
11	1	5ef00	2f7800	2ef000	1	0
12	1	6f780	37bc00	177800	1	0
13	0	37bc0	1bde00	0bbc00	1	0
14	0	1bde0	0def00	45de00	1	0
15	1	0def0	26f780	22ef00	0	0
16	1	06f78	337bc0	117780	1	0
17	1	037bc	39bde0	48bbc0	1	0
18	1	41bde	3cdef0	245de0	1	0
19	0	20def	1e6f78	122ef0	1	1
20	1	506f7	2f37bc	491778	1	1
21	0	2837b	179bde	248bbc	0	1

iter	key	LFSR1	LFSR2	LFSR3	maj	out
22	1	141bd	0bcdef	5245de	0	0
23	0	4a0de	05e6f7	2922ef	0	0
24	1	6506f	22f37b	149177	0	1
25	1	72837	3179bd	4a48bb	0	1
26	0	3941b	38bcde	25245d	1	0
27	0	1ca0d	3c5e6f	12922e	1	0
28	1	4e506	3e2f37	094917	1	0
29	0	27283	1f179b	44a48b	0	1
30	0	13941	0f8bcd	625245	1	1
31	0	49ca0	27c5e6	312922	0	0
32	1	24e50	13e2f3	189491	1	0
33	1	52728	29f179	4c4a48	0	1
34	1	29394	14f8bc	662524	0	0
35	1	149ca	2a7c5e	331292	1	0
36	0	4a4e5	353e2f	598949	1	1
37	0	65272	1a9f17	2cc4a4	0	1
38	1	72939	2d4f8b	566252	0	0
39	1	7949c	36a7c5	2b3129	1	0
40	0	7ca4e	3b53e2	559894	0	0
41	1	7e527	1da9f1	6acc4a	1	0
42	0	3f293	2ed4f8	356625	0	0

iter	key	LFSR1	LFSR2	LFSR3	maj	out
43	1	5f949	376a7c	5ab312	1	1
44	0	6fca4	1bb53e	2d5989	1	1
45	0	37e52	2dda9f	56acc4	1	1
46	0	5bf29	16ed4f	2b5662	1	0
47	1	6df94	2b76a7	15ab31	0	0
48	0	76fca	15bb53	0ad598	1	1
49	1	3b7e5	2adda9	056acc	1	0
50	1	1dbf2	156ed4	02b566	1	0
51	0	0edf9	0ab76a	415ab3	1	0
52	0	076fc	255bb5	20ad59	1	0
53	0	03b7e	32adda	1056ac	1	0
54	1	01dbf	1956ed	082b56	0	0
55	0	00edf	2cab76	0415ab	1	0
56	0	4076f	3655bb	020ad5	0	1
57	1	603b7	3b2add	41056a	0	0
58	0	301db	3d956e	6082b5	0	0
59	0	180ed	3ecab7	70415a	0	0
60	0	4c076	1f655b	7820ad	0	0
61	0	6603b	0fb2ad	3c1056	0	0
62	0	7301d	27d956	1e082b	0	0
63	0	3980e	33ecab	0f0415	0	0
64	0	1cc07	19f655	07820a	0	0

A5/1 Frame number insertion

iter	frame	LFSR1	LFSR2	LFSR3	maj	out
1	0	4e603	2cfb2a	03c105	1	0
2	1	67301	167d95	01e082	0	0
3	0	73980	2b3eca	00f041	1	1
4	1	79cc0	159f65	407820	1	1
5	1	7ce60	0acfb2	603c10	1	0
6	0	7e730	2567d9	301e08	1	1
7	1	3f398	12b3ec	580f04	0	0
8	0	1f9cc	0959f6	6c0782	0	0
9	1	0fce6	04acfb	3603c1	1	0
10	0	47e73	02567d	5b01e0	0	0
11	0	63f39	212b3e	2d80f0	1	1
12	1	71f9c	10959f	16c078	0	1
13	0	78fce	084acf	4b603c	1	1
14	1	7c7e7	242567	25b01e	1	0
15	1	7e3f3	3212b3	12d80f	0	1
16	0	7f1f9	190959	096c07	0	1
17	1	7f8fc	0c84ac	04b603	0	1
18	1	7fc7e	264256	025b01	1	1
19	1	3fe3f	13212b	012d80	0	0
20	1	5ff1f	299095	4096c0	1	0
21	0	6ff8f	34c84a	604b60	1	1
22	1	37fc7	1a6425	7025b0	0	0

A5/1 Key mixing (1)

iter	LFSR1	LFSR2	LFSR3	maj	out
1	37fc7	2d3212	3812d8	1	1
2	5bfe3	2d3212	1c096c	0	1
3	5bfe3	369909	4e04b6	1	0
4	6dff1	3b4c84	4e04b6	1	1
5	36ff8	1da642	4e04b6	0	0
6	36ff8	2ed321	27025b	0	0
7	36ff8	376990	13812d	1	1
8	5b7fc	1bb4c8	13812d	0	1
9	5b7fc	0dda64	49c096	1	0
10	2dbfe	06ed32	49c096	0	0
11	56dff	06ed32	64e04b	1	0
12	2b6ff	237699	64e04b	0	1
13	2b6ff	31bb4c	727025	1	0
14	15b7f	18dda6	393812	1	1
15	15b7f	2c6ed3	5c9c09	1	1
16	15b7f	163769	2e4e04	0	0
17	0adbf	2b1bb4	572702	1	1
18	056df	158dda	572702	1	1
19	42b6f	2ac6ed	572702	0	0
20	215b7	356376	6b9381	1	0

iter	LFSR1	LFSR2	LFSR3	maj	out
21	10adb	356376	35c9c0	0	1
22	0856d	3ab1bb	5ae4e0	0	0
23	0856d	1d58dd	6d7270	1	0
24	442b6	2eac6e	36b938	1	0
25	442b6	375637	5b5c9c	0	1
26	6215b	1bab1b	5b5c9c	1	0
27	6215b	0dd58d	6dae4e	0	0
28	310ad	26eac6	76d727	1	0
29	310ad	337563	3b6b93	0	1
30	58856	19bab1	1db5c9	1	0
31	58856	2cdd58	0edae4	1	0
32	58856	166eac	076d72	0	0
33	2c42b	166eac	43b6b9	1	0
34	56215	0b3756	21db5c	0	1
35	2b10a	259bab	21db5c	1	1
36	2b10a	12cdd5	10edae	0	1
37	55885	12cdd5	4876d7	1	1
38	55885	2966ea	643b6b	0	0
39	2ac42	34b375	643b6b	1	0
40	55621	34b375	321db5	1	1

iter	LFSR1	LFSR2	LFSR3	maj	out
41	2ab10	34b375	190eda	0	1
42	15588	3a59ba	4c876d	1	1
43	0aac4	3d2cdd	4c876d	0	0
44	45562	3d2cdd	6643b6	1	1
45	22ab1	3e966e	6643b6	0	1
46	11558	3f4b37	3321db	1	0
47	08aac	1fa59b	3321db	0	0
48	04556	0fd2cd	1990ed	1	0
49	022ab	0fd2cd	4cc876	0	0
50	41155	27e966	66643b	0	0
51	208aa	27e966	33321d	1	1
52	208aa	33f4b3	19990e	0	1
53	10455	19fa59	19990e	1	0
54	0822a	2cfd2c	4ccc87	0	1
55	04115	2cfd2c	266643	0	0
56	0208a	2cfd2c	133321	1	1
57	0208a	167e96	499990	1	0
58	0208a	2b3f4b	64ccc8	0	1
59	41045	2b3f4b	726664	0	0
60	20822	2b3f4b	793332	1	1

A5/1 Key mixing (2)

iter	LFSR1	LFSR2	LFSR3	maj	out
61	20822	159fa5	7c9999	1	0
62	20822	2acfd2	3e4ccc	0	0
63	10411	2acfd2	5f2666	1	1
64	48208	3567e9	5f2666	0	1
65	24104	3ab3f4	2f9333	0	1
66	52082	1d59fa	2f9333	1	1
67	52082	2eacfd	57c999	0	0
68	69041	2eacfd	2be4cc	0	0
69	74820	2eacfd	15f266	1	1
70	74820	37567e	4af933	0	1
71	7a410	3bab3f	4af933	1	0
72	3d208	1dd59f	657c99	0	0
73	1e904	0eeacf	657c99	1	0
74	1e904	077567	72be4c	0	1
75	4f482	03bab3	72be4c	1	1
76	67a41	01dd59	395f26	1	0
77	67a41	20eeac	1caf93	0	0
78	73d20	20eeac	4e57c9	1	1
79	79e90	107756	672be4	0	0
80	79e90	283bab	7395f2	1	1

iter	LFSR1	LFSR2	LFSR3	maj	out
81	3cf48	141dd5	39caf9	1	0
82	1e7a4	2a0eea	39caf9	1	1
83	0f3d2	350775	39caf9	0	0
84	479e9	3a83ba	1ce57c	0	1
85	23cf4	3d41dd	0e72be	1	1
86	11e7a	3d41dd	07395f	1	0
87	08f3d	3d41dd	439caf	1	1
88	4479e	3d41dd	21ce57	0	0
89	4479e	3ea0ee	10e72b	0	1
90	4479e	3f5077	487395	1	0
91	223cf	3f5077	2439ca	0	0
92	511e7	1fa83b	2439ca	1	0
93	511e7	0fd41d	521ce5	0	1
94	288f3	27ea0e	521ce5	1	0
95	288f3	33f507	290e72	0	0
96	54479	19fa83	548739	1	1
97	2a23c	0cfd41	548739	0	0
98	1511e	0cfd41	2a439c	0	1
99	0a88f	0cfd41	5521ce	0	0
100	45447	0cfd41	2a90e7	1	1

Task 3: Stream Cipher (2)

Stream Cipher (2)

Inputs:

- **key**: integer representing the shared secret key.
- **PRNG** (optional, default None): Iterator implementing a PRNG that produce a pseudorandom digit stream starting from an initial seed. A digit can be either a bit or a byte depending on the value of the digit parameter. If None an LFSR is used as PRNG with bit as digit.
- **digit**: (optional, default “bit”) string to determine the type of digit “bit” or “byte”.

Methods: **encrypt** and **decrypt** as before

Stream Cipher

Template:

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

  def __init__(self, key, prng=None, digit='bit'):
    ''' constructor docstring '''
    # do stuff
    self.prng = ...

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

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

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 40 to 256.

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

RC4

- Define a Iterator that implements the RC4 stream cipher.
- Given a key and a drop number, encrypt and decrypt a message.

```
message = 'hello world!’
key = b'0123456789ABCDEF'

# create a StreamCipher instance for both Alice and Bob
alice = StreamCipher(key, digit='byte', prng=RC4, drop=10)
bob = StreamCipher(key, digit='byte', prng=RC4, drop=10)

plaintextA = message.encode('utf-8')   # -> b'Hello world!'
ciphertext = alice.encrypt(plaintextA) # -> b'/\x9e\xf9\x83@\x81}\xa9\xd0\xd4\xd5\xf4'
plaintextB = bob.decrypt(ciphertext)   # -> b'Hello world!'
```