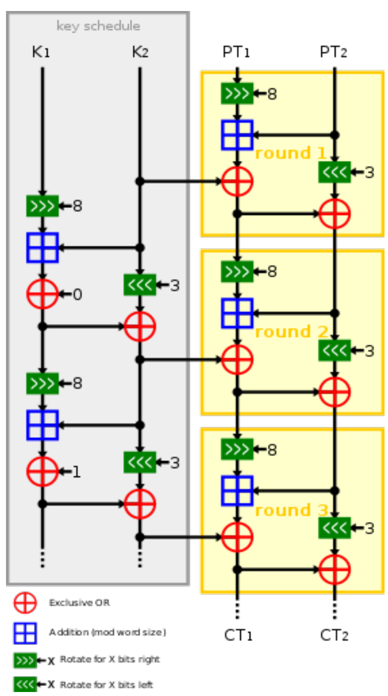
Crypto 3

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

Contents

3.1 Stream Ciphers: ARX . . . 2
3.1.1 RC4 . . . 3
3.1.2 Salsa20 . . . 6
3.1.3 Chacha20 . . . 9
3.2 **Bibliography** . . . 11



3.1 Stream Ciphers: ARX

ARX architecture consists of a long chain of three simple operations:

- addition $\boxplus$.
- exclusive-or, producing the xor $\oplus$.
- rotation, producing the rotation $<<<$ of constant number of bits to the left.

> On occasion I encounter the superstitious notion that these operations are too simple. In fact, these operations can easily simulate any circuit, and are therefore capable of reaching the same security level as any other selection of operations. The real question for the cipher designer is whether a different mix of operations could achieve the same security level at higher speed.
>
> **[Ber07, page 3]**

> The cryptographic strength of ARX comes from the fact that addition is not associative with rotation or XOR. However, it is very hard to estimate the security of such primitives.
>
> **Why Keccak is not ARX**

Here an example of non associativity of addition and XOR:
Consider the map $f : \mathbb{Z}_{2^n} \to \mathbb{Z}_{2^n}$ defined as

$$f(x, y) = x \boxplus y \ .$$

Then besides the simple expression the map f is not XOR-linear. Such XOR-linearity means that

$$f((x, y) \oplus (x', y')) = f(x, y) \oplus f(x', y')$$

but it is straightforward to check that:

$$f((1, 0) \oplus (0, 1)) \neq f(1, 0) \oplus f(0, 1) \ .$$

3.1.1 RC4

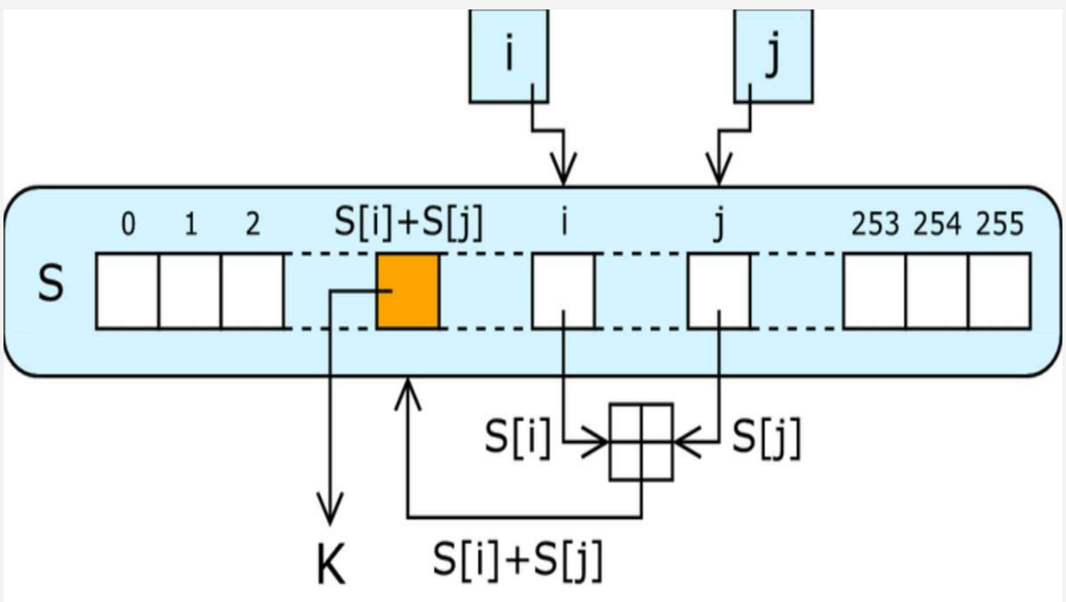
RC4 was designed by Ron Rivest of RSA Security in 1987.

RC4 & Internet

RC4 became part of some commonly used encryption protocols and standards, such as WEP in 1997 and WPA in 2003/2004 for wireless cards; and SSL in 1995 and its successor TLS in 1999, until it was prohibited for all versions of TLS by RFC 7465 in 2015, due to the RC4 attacks weakening or breaking RC4 used in SSL/TLS. The main factors in RC4's success over such a wide range of applications have been its speed and simplicity: efficient implementations in both software and hardware were very easy to develop.

RC4 has a register of 256 bytes and two pointers:

Figure 3.1.1: RC4



RC4 is initialized Init(k) with a key $k = k_1 k_2 \cdots k_l$ of keylenght $= l \leq 256$ bytes through the "Key Scheduling Algorithm" (KSA):

3.1.2 RC4: Init(k)

```
#
# Initizialization of RC4 or Key Schedule Algorithm (KSA)
#     input: a key = [k1,k2,....]  k_i numbers mod 256
#     output: register = [ , , ... , ] array of length 265
#             with numbers mod 256
#


from swap import swap

def Init(key):
    register = [i for i in range(0,256)]
    j=0
    l = len(key)
    for i in range(0,256):
        j = (j + register[i] + key[i%l])%256
        swap(register,i,j)
    return register
```

Exercise 3.1.3

Write the swap.py function. What is the state of the register after initialized with $k = [0]$?

The GetBits of RC4 is actually a ”get bytes”.

Here is a **python** implementation of the ciphering algorithm of RC4:

3.1.4 RC4 ciphering

```
# RC4:
#    input :
#            key = 'ASCII' string
#            Plaintext = 'ASCII' string
#
#    output:
#            ciphertext = array of hexadecimal

from swap import swap
from RC4KSA import Init

def RC4(key,Plaintext):
    register = Init(key)
    i=0
    j=0
    ciphertext=[]
    for r in range(0,len(Plaintext)):
        i = (i+1)%256
        j = (j + register[i])%256
        register = swap(register,i,j)
        cr = Plaintext[r]^(register[(register[i]+register[j])%256])
        ciphertext.append(cr)
    return ciphertext
```

RC4
https://en.wikipedia.org/wiki/RC4.

3.1.2 Salsa20

Salsa20 expands a 256-bit key and a 64-bit nonce into a 2^{70}-byte stream. It encrypts a b-byte plaintext by xoring the plaintext with the first b bytes of the stream and discarding the rest of the stream. It decrypts a b-byte ciphertext by xoring the ciphertext with the first b bytes of the stream.

Salsa20

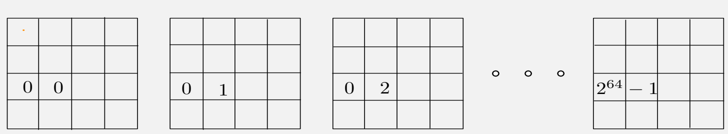
Here is the internal initial state of Salsa20:

Cons	Key	Key	Key
Key	Cons	Nonce	Nonce
Pos	Pos	Cons	Key
Key	Key	Key	Cons

It is 4×4 matrix of "words" of $32 = 2^5$ bits. So it has $2^5 \times 2^4 = 2^9$ bits. Along the diagonal Cons Cons Cons Cons is written "expand 32-byte k" in ASCII. The words Pos Pos, called positions are used to construct the stream flow of 2^{73} bits.
Here is how to construct the stream. First of all the entries of the 4×4 matrix are numerated as

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

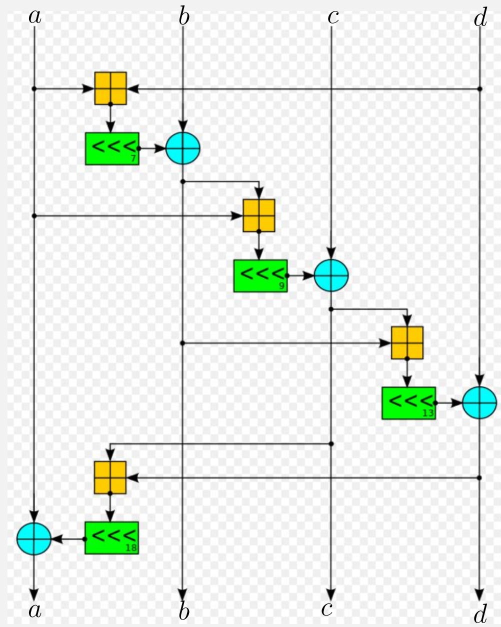
To encrypt a b-byte plaintext $\lceil \frac{b}{64} \rceil$ copies of the initial state are initialized using Pos Pos as a counter from 0 to $\lceil \frac{b}{64} \rceil - 1$.



Salsa20: quarter-round function

Each one of these states is scrambled 10 times each of 2 rounds using QR(a, b, c, d):

$$
\begin{aligned}
b &\oplus= (a + d) <<< 7 \\
c &\oplus= (b + a) <<< 9 \\
d &\oplus= (c + b) <<< 13 \\
a &\oplus= (d + c) <<< 18
\end{aligned}
$$



Here the 2 rounds :

QR(0,4,8,12)
QR(5,9,13,1)
QR(10,14,2,6)
QR(15,3,7,11)

QR(0,1,2,3)
QR(5,6,7,4)
QR(10,11,8,9)
QR(15,12,13,14)

The output is the ADD ($\boxplus$), in $\mathbb{Z}_{2^{32}}$, of the matrix after the 20 rounds with the initial matrix. **This last ADD is fundamental. Otherwise, since each round is invertible, without the ADD it is possible to go back to the initial matrix and recover the key.**

NOTE 3.1.5

`Salsa20` does not guarantee authenticity of the data you decrypt! In other words, an attacker may manipulate the data in transit. In order to prevent that, you must also use a *Message Authentication Code* to authenticate the ciphertext (*encrypt-then-mac*).

MAC = Message Authentication Code.

See e.g. https://en.wikipedia.org/wiki/Bit-flipping_attack.

3.1.3 Chacha20

Also ChaCha20 expands a key of 256 bits to a stream 2^{73} bits. The important difference between ChaCha and Salsa is the QR function. The improvement goal is to keep the speed performance but improving the diffusion of bits.

ChaCha20

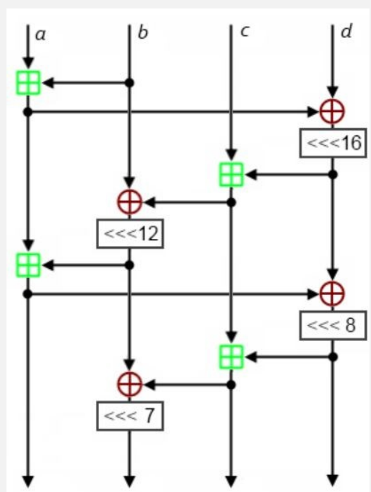
Here the ChaCha20 initial state:

Cons	Cons	Cons	Cons
Key	Key	Key	Key
Key	Key	Key	Key
Pos	Pos	Nonce	Nonce

Up to the position of the words is similar to that of Salsa20.

But the $\text{QR}(a, b, c, d)$ is quite different:

$$
\begin{aligned}
&a += b\\
&d \oplus = a\\
&d\ = d <<< 16\\
&c += d\\
&b \oplus = c\\
&b\ = b <<< 12\\
&a += b\\
&d \oplus = a\\
&d\ = d <<< 8\\
&c += d\\
&b \oplus = c\\
&b\ = b <<< 7
\end{aligned}
$$



ChaCha20:rounds

Here the 2 internal rounds:

QR(0, 4, 8, 12)
QR(1, 5, 9, 13)
QR(2, 6, 10, 14)
QR(3, 7, 11, 15)

QR(0, 5, 10, 15)
QR(1, 6, 11, 12)
QR(2, 7, 8, 13)
QR(3, 4, 9, 14)

As with Salsa20, the output is the ADD in $\mathbb{Z}_{2^{32}}$ of the matrix state after the 20 round with the initial one.

Exercise 3.1.6

Download the Salsa20 Pyhton implementation in `https://pypi.org/project/salsa20/` and follow the example.

Exercise 3.1.7

Read the introduction of `http://cr.yp.to/snuffle/salsafamily-20071225.pdf`.

3.2 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.

Here is the list of papers:

[Ber07] Daniel J. Bernstein; *The Salsa20 family of stream ciphers*, https://cr.yp.to/snuffle/salsafamily-20071225.pdf

[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:

https://crypto.stackexchange.com/questions/tagged/salsa20?sort=frequent&pageSize=50

https://en.wikipedia.org/wiki/Daniel_J._Bernstein

https://keccak.team/2017/not_arx.html