Block Ciphers

Elements of Applied Data Security M

Alex Marchioni – alex.marchioni@unibo.it

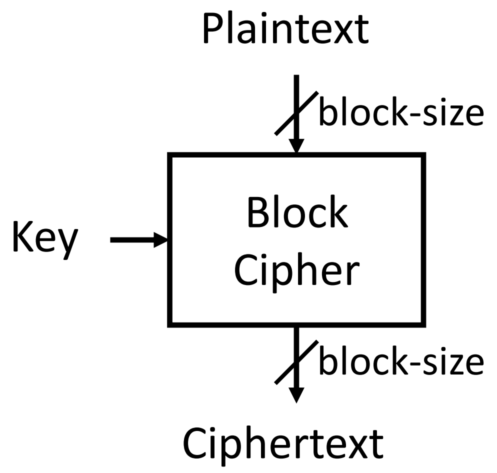
Livia Manovi – livia.manovi@unibo.it

Block Ciphers

Unlike Stream Ciphers, which encrypt one bit at a time, Block Ciphers encrypt a **block of text**.

- AES encrypts 128-bit blocks.

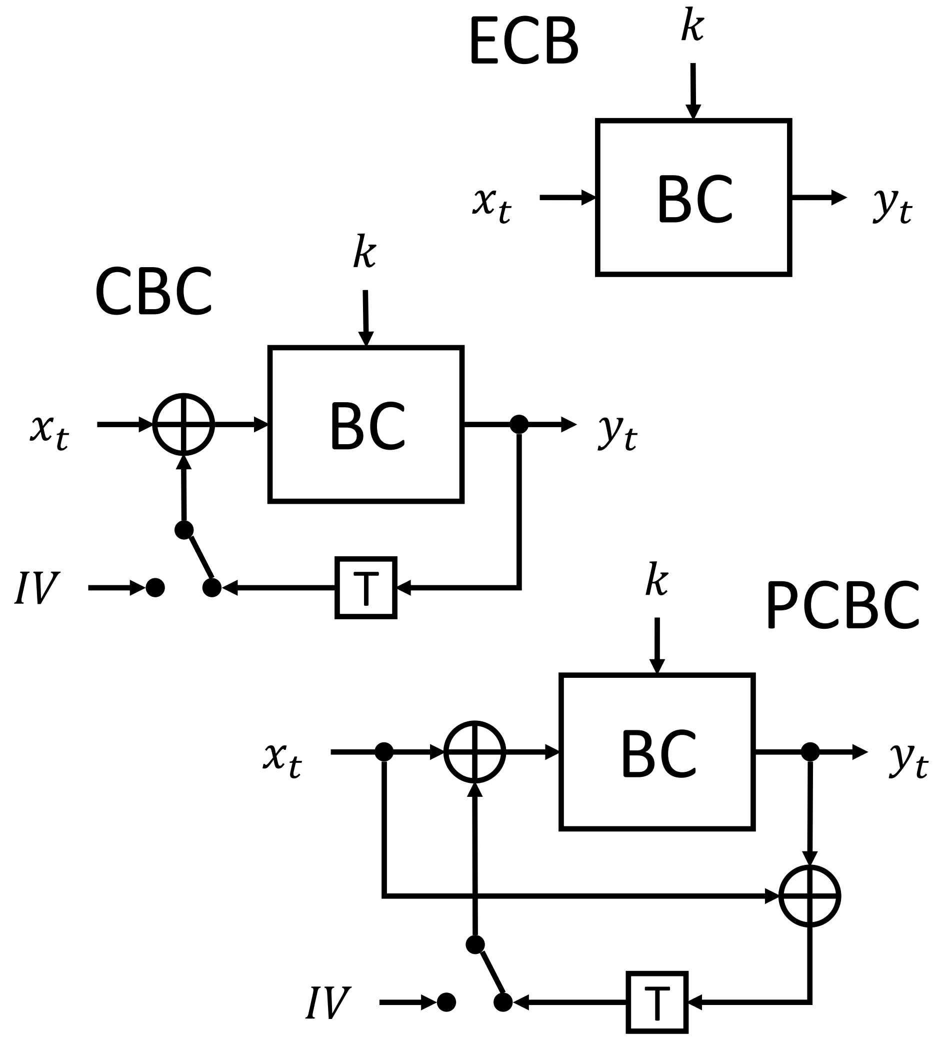
Since a block cipher is suitable only for the encryption of a single block under a fixed key, a multitude of **modes of operation** have been designed to allow their repeated use in a secure way.



Moreover, block ciphers may also feature as **building blocks** in other cryptographic protocols, such as universal hash functions and pseudo-random number generators.

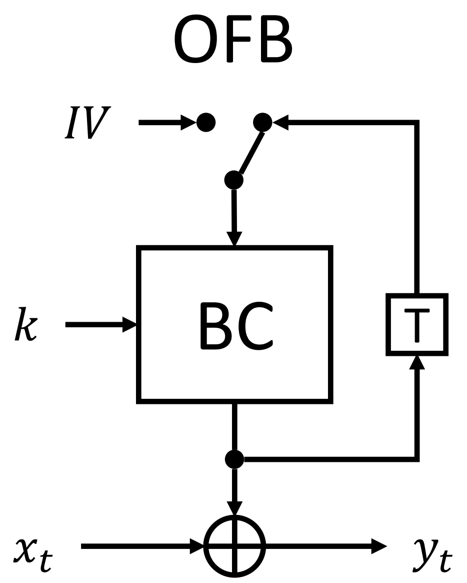
Modes of Operation

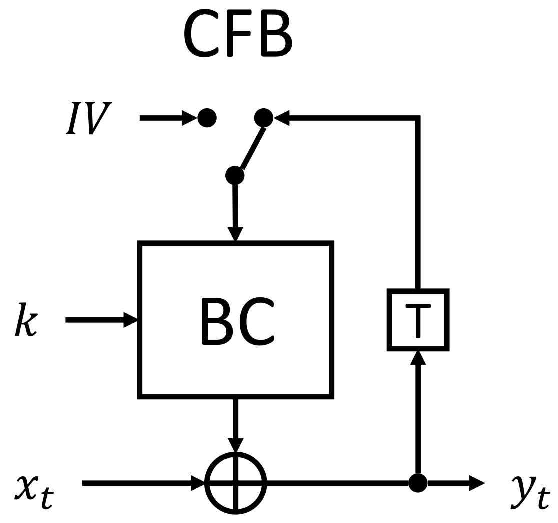
- **Electronic codebook (ECB)**: the simplest of the encryption modes where the message is divided into blocks, and each block is encrypted separately.
- **Cipher block chaining (CBC)**: each block of plaintext is XORed with the previous ciphertext block before being encrypted.
- **Propagating cipher block chaining (PCBC)**: each block of plaintext is XORed with both the previous plaintext block and the previous ciphertext block before being encrypted.

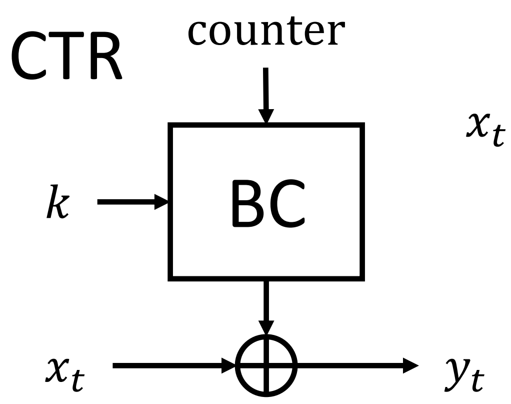


Modes of Operation

- **Output feedback (OFB)**: it generates keystream blocks, which are then XORed with the plaintext blocks to get the ciphertext.
- **Cipher feedback (CFB)**: similar to OFB, but to generate the new keystream block, it employs the previous ciphertext instead of the previous keystream block .
- **Counter (CTR)**: the keystream block is generated from the value of a counter that is incremented at each new block.







Tasks

1. AES
2. Monte Carlo Simulations
3. Diffusion and confusion with AES

Task 1: AES

Advanced Encryption Standard (AES)

AES is a specification for the symmetric-key encryption established by the NIST in 2001 and then adopted by the U.S. government.

The standard comprises three block ciphers from a larger collection originally published as **Rijndael**. Each of these ciphers has a 128-bit block size, with key sizes of 128, 192 and 256 bits.

[1] FIPS PUB 197, Advanced Encryption Standard (AES), National Institute of Standards and Technology, U.S. Department of Commerce, November 2001.

[2] Joan Daemen and Vincent Rijmen, The Design of Rijndael, AES - The Advanced Encryption Standard, Springer-Verlag 2002 (238 pp.)

Python Packages for Cryptography

Pycryptodome: self-contained Python package of low-level cryptographic primitives. It is a fork of PyCrypto that has been enhanced to add more implementations and fixes to the original library.

PyNaCl: Python binding to libsodium, which is a fork of the Networking and Cryptography library. These libraries have a stated goal of improving usability, security and speed.

Cryptography: cryptography is a package which provides cryptographic recipes and primitives to Python developers. It includes both high level recipes and low-level interfaces to common cryptographic algorithms such as symmetric ciphers, message digests, and key derivation functions.

Pycryptodome

PyCryptodome is a self-contained Python package of low-level cryptographic primitives. It is organized in sub-packages dedicated to solving a specific class of problems:

- `Crypto.Cipher`: Modules for protecting **confidentiality** that is, for encrypting and decrypting data (example: AES).
- `Crypto.Signature`: Modules for assuring **authenticity**, that is, for creating and verifying digital signatures of messages (example: PKCS#1)
- `Crypto.Hash`: Modules for creating cryptographic **digests** (example: SHA-256).
- `Crypto.PublicKey`: Modules for generating, exporting or importing public keys (example: RSA or ECC).

`Crypto.Cipher` subpackage

The base API of a cipher is fairly simple:

- You instantiate a cipher object by calling the `new()` function from the relevant cipher module. The first parameter is always the cryptographic **key**. You can (and sometimes must) pass additional cipher- or mode-specific parameters such as a nonce or a mode of operation.

```
from Crypto.Cipher import AES

key = b'0123456701234567’
cipher = AES.new(key, AES.MODE_ECB)
```

`Crypto.Cipher` subpackage

- For encrypting data, you call the `encrypt()` method of the cipher object with the plaintext. The method returns the piece of ciphertext. For most algorithms, you may call encrypt() multiple times (i.e. once for each piece of plaintext).
- For decrypting data, you call the `decrypt()` method of the cipher object with the ciphertext. The method returns the piece of plaintext.

```
plaintextA = b'this is a secret'
ciphertext = cipher.encrypt(plaintextA) # b'\x8dk\x84\xcey*h\xach\x9b\xd0[\xb6pR\x95'
plaintextB = cipher.decrypt(ciphertext) # b'this is a secret'
```

Task 1 – AES

- Create an instance of the AES class from PyCryptodome for each of the following operation modes: ECB, CBC, CFB, CTR.
  - When needed you can use a null or random initialization vector (IV)
- For each instance of AES,
  - transform the image into a byte sequence
  - encrypt the sequence
  - display the ciphertext as an image
  - comment your results.

Task 1 – AES

- **Input**
  - Plain image `image.png`
- **Outputs**
  - For each AES mode of operation:
    - Encrypted image `ciphertext_mode=<mode>.png`
    - Comment on the results

Dealing with images

- Load .png image

```python
from matplotlib.image import imread

image_float = imread('image.png’).mean(axis=-1)
image = 255 * image_float.astype(np.uint8)
```

- Turn image into a plaintext, encrypt it, and turn the ciphertext into an image

plain image

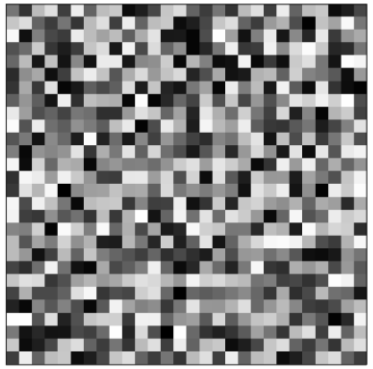
```
b'\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x04\x0f\x05\x05\x
14\x19!%%%\x07\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x1e\x8c\x80\x81\xaa\xb8
\xcd\xd9\xd9\xd9\xca_\x14\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x002\xdb\xfa\xfa\xfc
\xfd\xfe\xfe\xfe\xfe\xfd\xe7{\n\x0
0\x00\x00\x00\x00\x00\x00' ...
```

plain text

$\rightarrow \boxed{\text{AES}} \rightarrow$

```
b'u4\xbb\xdf>\xedc\xbd\xc8\x0c\x1e
G\xc6\xddc=u4\xbb\xdf>\xedc\xbd\xc
8\x0c\x1eG\xc6\xddc=u4\xbb\xdf>\xe
dc\xbd\xc8\x0c\x1eG\xc6\xddc=\x9a\
xb2M\x02\xc2\xa4\xf9w\xf8\x10\x7fH
;\xeb\xbc\xa2\xbc_\xac\xe4\x18\x19
\x05M\xfe\xa0\x12\x80\xeb\x17\x89&
\xe9:\x95\xaaRl\xcb6\x1d<\xc5\xdc\
x17\xf1J\xdbe\x8e\x05'w\xef7<\xf1\
x8b\xdb\xfb\xc8&,ug\xbb\x16\xc1\x1
4\x88\x12\xeb\xeb\x17\x88\x8f\xcc\
x08\xb6\x81\x0c\x11\xf4\x08:\x0f\x
ca\xdfw\x14\xfd\xf5' ...
```

cipher text



cipher image

Task 2: Monte Carlo Simulations

Monte Carlo Simulations (MCS)

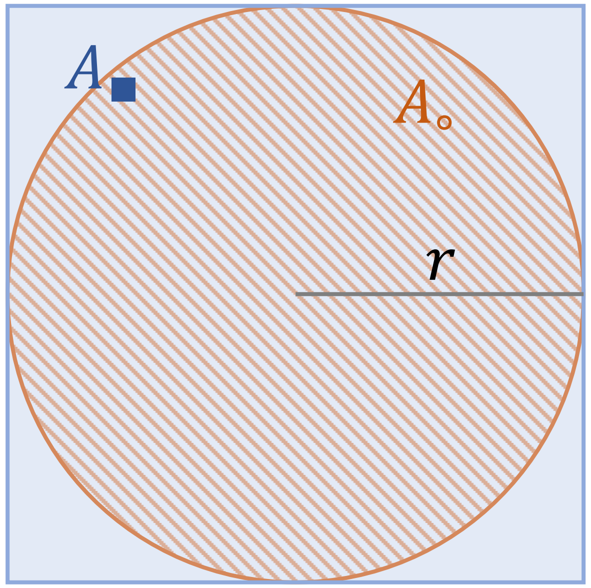
Monte Carlo simulations is a statistical technique that allows for the modelling of complex situations where many random variables are involved. It uses the process of repeated random sampling to model stochastic systems and determine the odds for a variety of outcomes.

The idea comes from the Law of Large Numbers that states: *the average of the results obtained from a large number of trials should be close to the expected value and will tend to become closer to the expected value as more trials are performed.*

Roughly speaking, If you do not know some parameters of your system, you can make several trials and then take the average.

Estimating π with MCS

A classical example is the estimation of π.



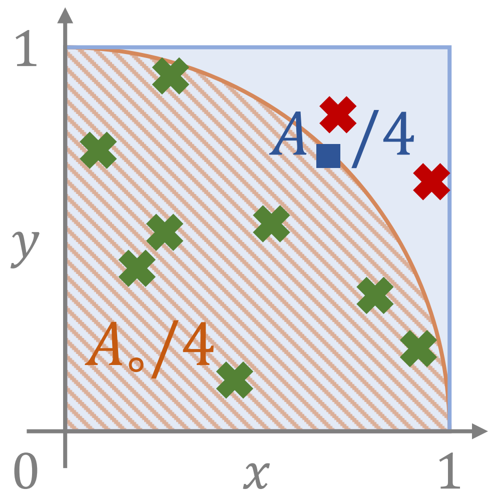
$$\frac{A_\circ}{A_\blacksquare} = \frac{\pi r^2}{(2r)^2} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4} \quad \Rightarrow \quad \pi = 4\frac{A_\circ}{A_\blacksquare}$$

The ratio $A_\circ / A_\blacksquare$ can be estimated by considering random points uniformly distributed in the square and counting how many are in the circle with respect to the total.

Estimating π with MCS

Let us consider a quarter of a square and a circle with $r = 1$.

We can draw N coordinates (x, y) as instances of uniform random variables $x, y \in U([0,1])$ and count how many fall into the circle with respect to the total.

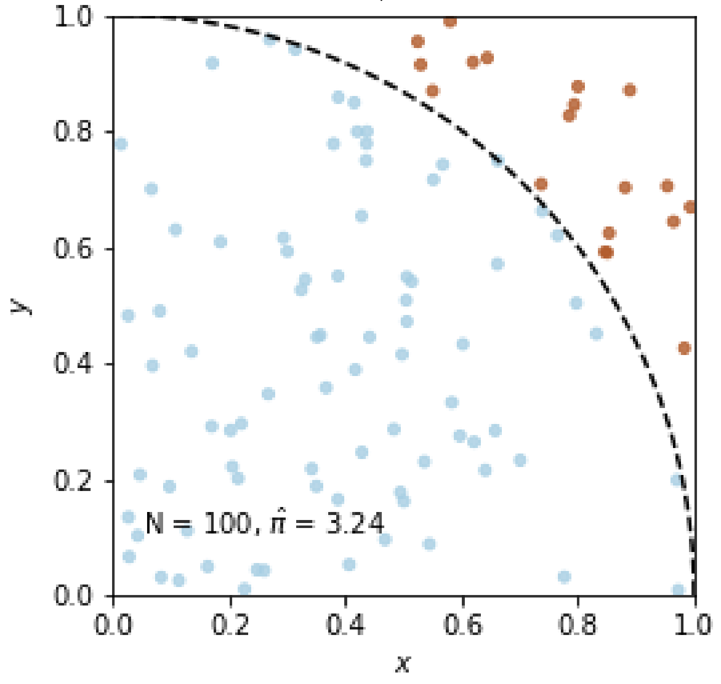


$$\pi = 4\frac{A_\circ/4}{A_\blacksquare/4} \sim 4\frac{\#inside}{\#total} = 4\frac{8}{8+2} = 3.2 = \hat{\pi}$$

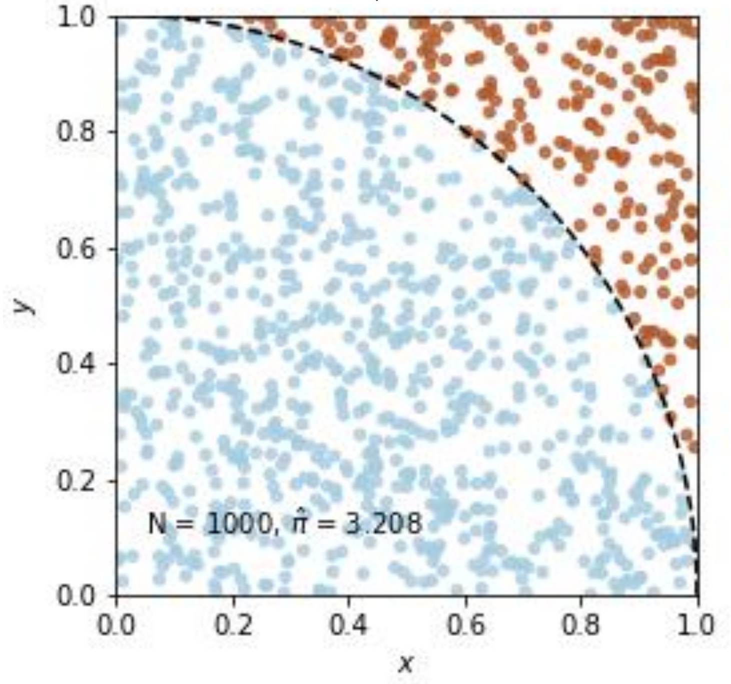
$$\text{err} = |\pi - \hat{\pi}| \sim 0.0584, \qquad \text{err}_{\%} = \frac{|\pi - \hat{\pi}|}{\pi} \sim 1.86\%$$

Estimating π with MCS

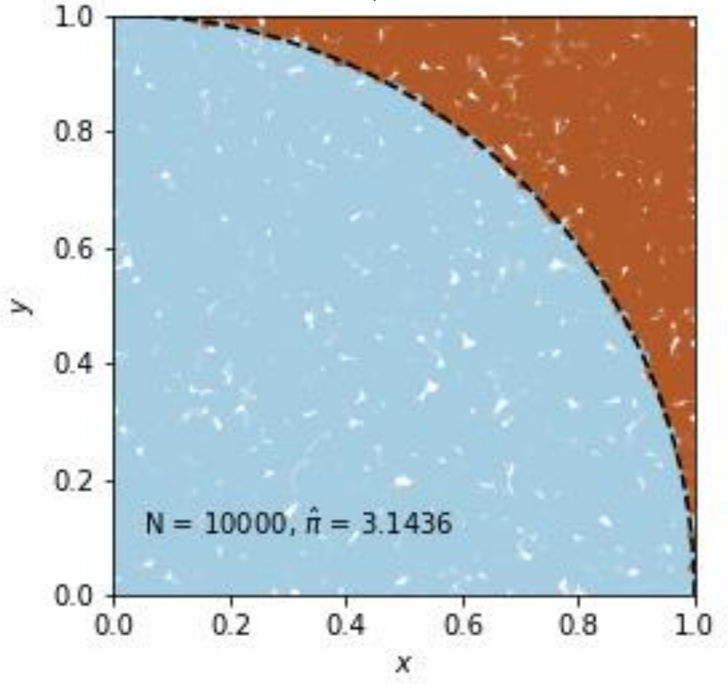
$N = 100, \hat{\pi} = 3.24$



$N = 1000, \hat{\pi} = 3.208$



$N = 10000, \hat{\pi} = 3.1436$



Task 2

- Estimate the value of π by means of MonteCarlo Simulations as explained in the previous slides.

- Discuss the choice of the number of trials for the estimate to be accurate enough.
  - How does the estimation error approach zero?
  - If I did not know the actual value of π, how can one decide when to stop?

Task 3: AES Diffusion and Confusion

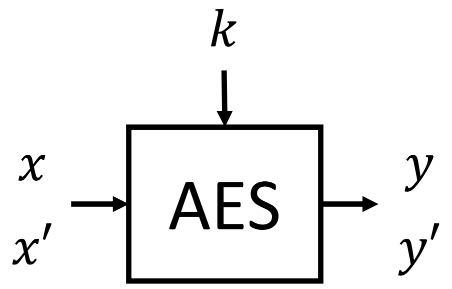
Diffusion with MCS

Diffusion: *if we change a single bit of the plaintext, then (statistically) half of the bits in the ciphertext should change*.

We want to test qualitatively if AES provides diffusion.

We randomly draw a plaintext x and a key k, change a bit in the plaintext and then observe how this change affects the ciphertext.

Plaintext x' differs only 1bit from plaintext x.



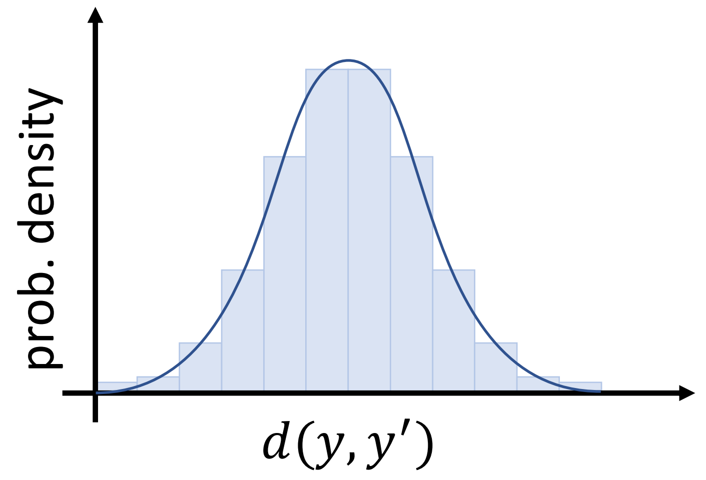
How many bits does the ciphertext y' differ from the ciphertext y?

Diffusion with MCS

We measures the difference between y and y' as the number of bits for which y' differs from y – the Hamming distance $d(y, y')$.

We are interested in finding the **distribution** of such distance, which we can estimate with Monte Carlo Simulations.

We do not expect to find that $d(y, y')$ is always $n/2$ (where n is the number of bits in the ciphertext) but we expect to find a distribution that is concentrated around $n/2$.

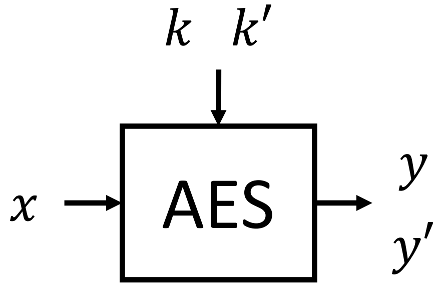


Confusion with MCS

Confusion: *the relationship between the key and the ciphertext must be very complicated and impossible (hard) to invert*

At bit level, it means that each bit of the ciphertext depends on several parts of the key. As a result, if a single bit is changed in the key, many bits in the ciphertext change.

Again, we want to assess confusion of AES qualitatively.



The keys k and k' differ only by 1bit.
What is the distribution of the Hamming distance between the ciphertext y and y' obtained by encrypting the same plaintext x with the two different keys k and k'?

Task 3

- Select the most appropriate Mode of Operation to characterize the AES block cipher in this kind of analysis.
- Assess diffusion and confusion and answer the following questions:
    - How does the distribution of Hamming distances change by varying the length of the key?
    - Are there any differences between diffusion and confusion?

Bonus Task: RC4 Diffusion and Confusion

Rivest Cipher 4 (RC4)

RC4 is a **stream cipher** that generates a **stream of bytes** from a secret internal state which consists of two components:

- 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.

The stream of pseudo-random bytes is then XORed with the plaintext to produce the ciphertext.

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 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**
$i, j \leftarrow 0, 0$
**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 provides an output byte.

```
**State** $P, i, j$
$i \leftarrow (i + 1) \bmod 256$
$j \leftarrow (j + P[i]) \bmod 256$
$P[i], P[j] \leftarrow P[j], P[i]$
$B \leftarrow P[(P[i] + P[j]) \bmod 256]$
**Output** $B$
```

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: RC4

- Define an Iterable that implements the RC4-based stream cipher, that given a key and an optional drop number, encrypt and decrypt a message (bytes).

```
plaintextA = b'hello world!'
key = b'0123456789ABCDEF’
n = 768

# create an instance of the RC4-based stream cipher
alice = RC4(key, drop=n)
bob = RC4(key, drop=n)

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

- Evaluate the properties of diffusion and confusion with MonteCarlo Simulations.

Deadline

Tuesday, May 7th at 12PM (noon)