🔐 Tutorial 4: Symmetric Encryption

In this tutorial, we will explore some foundational concepts in **historical cryptographic schemes** and **symmetric cryptography**.
Recall that in **symmetric key encryption**, the same secret key is used for both the encryption and decryption processes.

🧩 1. Revisiting Simple Encryption

You are given a text file named **enc.txt** by a friend. They say they have no idea what it means—after all, they've never taken a class in cryptography (and they didn't enjoy history much in school either).

They watch you in wonder and awe as you begin decrypting it using Python, without relying on any cryptography libraries.

Your task is to **decrypt the given file `enc.txt`**.

> **Recall:** This is a *ciphertext-only attack* scenario, where you are given only the encrypted message (ciphertext) and no information about the plaintext or the key. Your job is to analyze the ciphertext and deduce the plaintext through cryptanalysis techniques.

a) Observe the contents of the given file

Write code that prints the contents of the file.

In [ ]:
```
file =          # TODO: Open the enc.txt file
content =       # TODO: Read the contents
# TODO: Print the contents
# Close the file
```

b) Check the properties of the ciphertext

That doesn't reveal much, does it?
Let's examine some properties of the ciphertext:

- Check the length of the ciphertext.
- Find the factors of its length (use the **factorint** function in **sympy**) – Reference
- Which numbers divide the length?

In [ ]:
```
from sympy.ntheory import factorint

## TODO print the total length of the content
## TODO use the factorint function to find the prime factors of content l
```

Still not much insight. Let's perform some alphabet/symbol analysis:

- Generate frequency plots assuming symbol lengths from 1 to 5.
- Write the code in a generic way:
  - Allow input of the symbol length to consider.
  - The code should progressively identify symbols, store them in a dictionary, and track each symbol's frequency.
  - Once the dictionary is fully built, plot the frequency histogram of the symbols.
  - Based on the plots, which symbol length do you think is the correct one?

In [ ]:
```
## TODO: Write a function to build the symbol frequency dictionary
## TODO: It will take the filename and the symbol length as inputs

def build_symbol_dictionary(filename, symbol_length):
    file = ## TODO: Open the given file
    count = ## TODO: Build an empty dictionary
    symbol = ""                 ## Placeholder for the symbols - no need

    while True:
        # TODO: Read the file by symbol_length
        symbol =

        if not symbol:          ## TODO: If there is no symbol, it means


        else:
            if symbol in count: ## TODO: If the current symbol is already

            else:               ## TODO: Else, add the symbol to the dicti
```

In [ ]:
```
symbol_frequency = build_symbol_dictionary("enc.txt", 3)

print(symbol_frequency.keys())
print(sum(symbol_frequency.values()))
```

In [ ]:
```
import matplotlib.pyplot as plt

plt.bar(list(symbol_frequency.keys()), symbol_frequency.values(), color='
plt.xticks(rotation = 45, size =6)
plt.xlabel("Symbol")
plt.ylabel("Frequency")
plt.show()
```

c) Decide on the symbol length

From the above analysis decide on the symbol length. Now we need to try some substitutions

- What symbol can be the whitespace?
- What symbol can be the letter 'e'?
- Write a code to try out a few substitutions and see what makes sense. Try several runs with different guesses.

In [ ]:
```
file = open("enc.txt", 'r')

content=file.read()
sub_content=""

for i in range(0,len(content),3):
    symbol = content[i:i+3]
    if symbol== ##TODO Experiment with guessing some symbols. You can add
        symbol=
    if symbol== ##TODO Experiment with guessing some symbols. You can add
        symbol=
    sub_content = sub_content+symbol
print(sub_content)

file.close()
```

Hopefully by now you have identified some correct mappings. You can continue this process to find all the mappings, but it can be a bit tedious. Let's see if we can identify a pattern:

- Build a new sorted dictionary.
- Replot the symbol histogram (this time sorting the keys).

In [ ]:
```
keys = list(symbol_frequency.keys())
keys.sort()
sorted_dict = {i: symbol_frequency[i] for i in keys}

import matplotlib.pyplot as plt

plt.bar(list(sorted_dict.keys()), sorted_dict.values(), color='b')
plt.xticks(rotation = 45, size =6)
plt.xlabel("Symbol")
plt.ylabel("Frequency")
plt.show()
```

Based on this graph and the substitutions you found earlier, you can now express the mapping as a formula.

- What is the equation for the mapping?

d) Write a code to decipher the full text.

In [ ]:
```
import string

chars = list(string.ascii_lowercase)

file = open("enc.txt", 'r')

content = file.read()
file.close()

plain_content = ""

for i in range(0, len(content), 3):
    symbol = content[i:i+3]
    if symbol !=  # TODO: Write the program logic for non-whitespace char
```

```
    else:
        plain_content += " "

print(plain_content)

with open("plain.txt", "w") as text_file:
    n = text_file.write(plain_content)
```

🔑 2. Doing it Right — Encryption

Once you have decrypted `enc.txt`, you decide to do it properly this time.

- Write code to read the decoded text file and encrypt it using AES-CBC.
- Use the `pycrypto` library: https://pycryptodome.readthedocs.io/en/latest.
- You might need to pad your data. Why?
- Choose a key length of 128 bits.
- Generate an IV using `Crypto.Random`.
- Print the output as a hex-encoded string.
- Write the output as binary to `text.aes`.

In [ ]:
```
# from os import lseek  # (Unused import)
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad

def AES_Encrypt(inFileName, outFileName):
    # TODO: Generate random 16-byte key and IV
    key =
    iv =

    cipher = AES.new(key, AES.MODE_CBC, iv)

    # TODO: Load the input file
    with open(inFileName, 'rb') as text_file:
        plaintext =

    # TODO: Perform the encryption
    ciphertext =

    # TODO: Store the encrypted file
    with open(outFileName, "wb") as cipher_file:
        cipher_file.write(ciphertext)

    # TODO: Return the key and initialization vector (needed for decrypti
    return key, iv

key, iv = AES_Encrypt("plain.txt", "testing.enc")
```

🔑 3. Doing it Right — Decryption

Write a code snippet to check whether you can decrypt the file you just encrypted.

In [ ]:
```python
from Crypto.Util.Padding import unpad

def AES_Decrypt(key, iv, inFileName):

    # TODO: Load the encrypted file
    # TODO: Perform the decryption


    print(plaintext)

AES_Decrypt(key, iv, "testing.enc")
```

🕵️ 4. Ciphertext Tampering in CBC Mode

AES in CBC (Cipher Block Chaining) mode links each ciphertext block to the previous one. See the given figure below.

Now consider the following:

You have a ciphertext encrypted using AES-CBC.

One byte of a ciphertext block is corrupted (e.g., flipped or modified).

Questions:

1. What will happen to the decryption of the corrupted block?
2. What will happen to the next block during decryption?
3. Will the remaining blocks (after the next one) decrypt correctly or incorrectly? Why?