醫學圖像加密的研究與應用

第 14 組

組長： 4109064217_黃梓誠

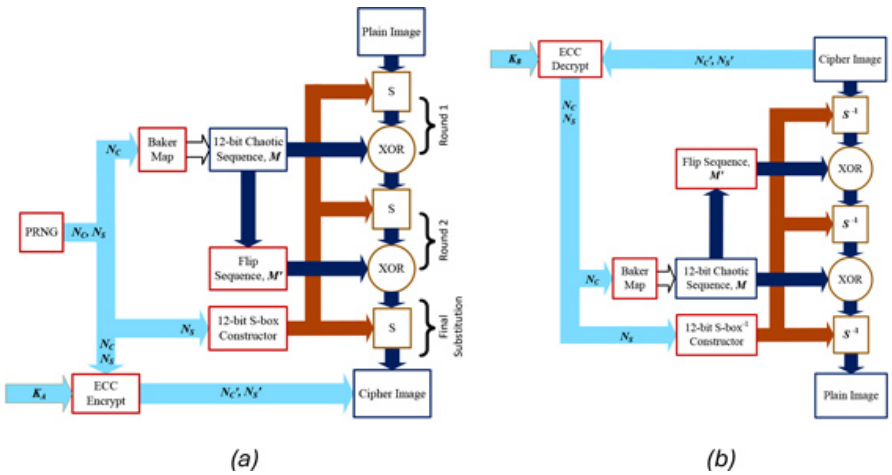
組員： 4109064208_方崇瑋、4109064210_游宸睿、4109064233_李文弘

動機與簡介:

隨著科技發展與醫學進步，醫學圖像逐漸成為醫療領域中不可或缺的一部分，醫學圖像包含敏感的醫療信息，例如病人的病歷、診斷結果等，因此需要有效的加密方案，專門用於保護醫學圖像的安全性。不過由於醫學影像(12-bit)在精確度上較傳統的 8-bit 影像高，為了不影響醫學影像的精確度，無法直接將傳統的 8-bit 的影像加密方式直接應用在醫學影像中。即使將 12-bit 的醫學影像轉換至 8-bit 再進行影像加密也可能會影響加密方法的安全性、品質以及效率。

- 這篇論文的目標是：
  1. 保留醫學影像訊息的準確度，不將 12-bit 的影像像素降為 8-bit。
  2. 提高加密安全性及有效性。
  3. 改善加密效率。

本論文提出動態 12*12 bit S-box 進行加密，其中 S-box 將一組 12-bit 串列作為輸入，並且透過非線性的方式將其轉換為一組 12-bit 的輸出，並且透過將傳統的 chaotic map 從 8-bit 改為 12-bit 以增強此加密方式的安全性，同時也驗證 12-bit chaotic map 方法的有效性。



所選論文:

https://ieeexplore.ieee.org/abstract/document/10460526

```
@article{ibrahim2024new,
 title={A New 12-Bit Chaotic Image Encryption Scheme Using A 12$\times$ 12 Dynamic S-Box},
 author={Ibrahim, Saleh and Abbas, Alaa and Alharbi, Ayman and Albahar, Marwan},
 journal={IEEE Access},
 year={2024},
 publisher={IEEE}
}
```

程式實作:

- 本篇論文未提供相關程式碼，所有 Code 皆由我們撰寫
- 我們實作的 Github： https://github.com/Rui0828/2024-IS-Self-Learning-Project.git
- 程式說明至於附錄

實驗結果分析:

實驗結果的部分我們進行了四種類型的實驗結果分析，分別為統計分析(Statistical Analysis)、密鑰敏感度(Key Sensitivity)、抗選擇明文攻擊能力(Resistance to Chosen Plaintext Attacks)、密鑰空間分析(Key Space Analysis)。這四種分析主要在最這個加解密方式進行不同層面的安全性分析，旨在分析其有效性以及可靠性。

1. **統計分析 (Statistical Analysis)**

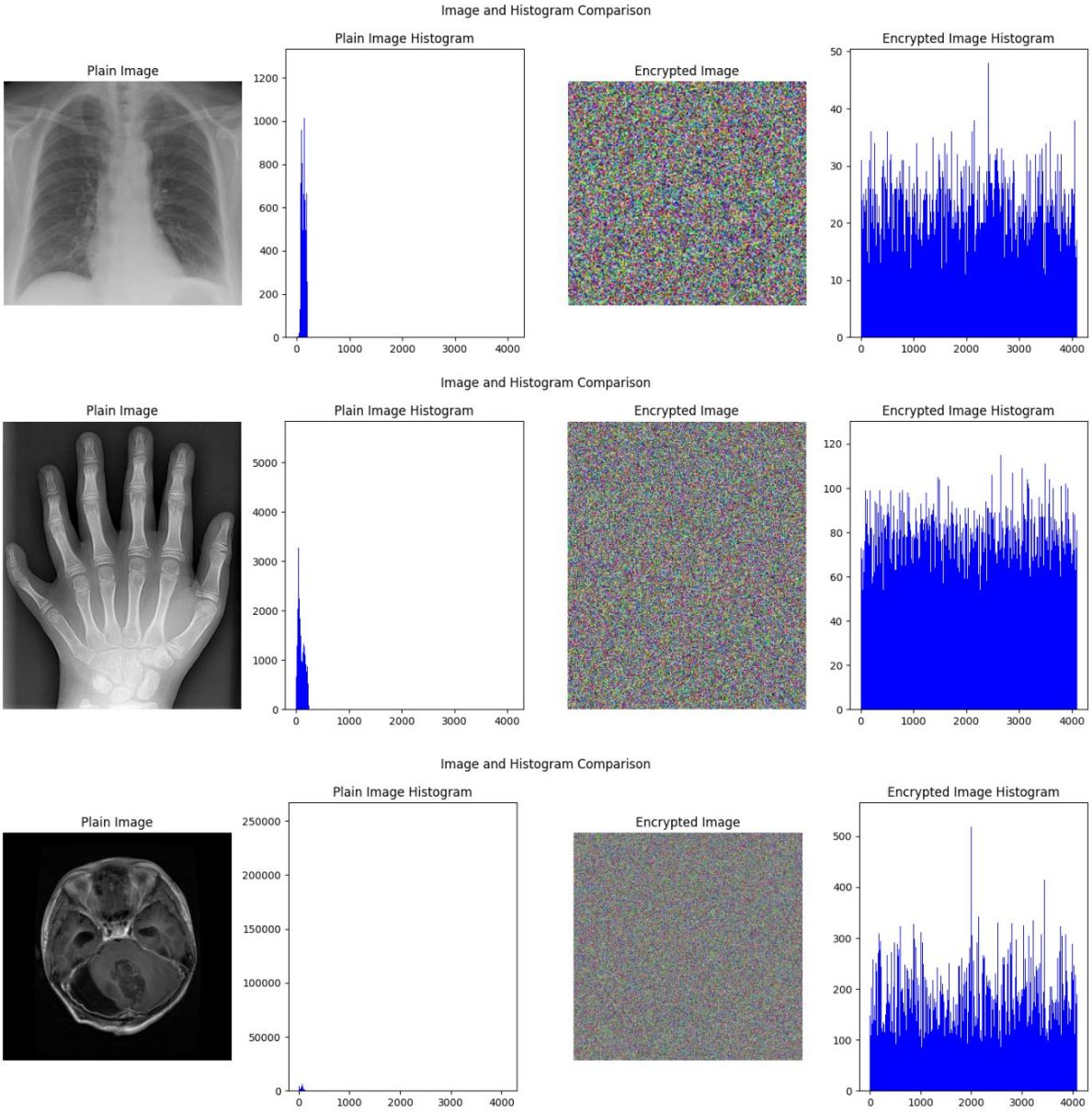
   此部分是透過分析每個像素的出現頻率、出現的隨機性等來對圖像抵禦抵抗統計型攻擊 (statistical attacks) 的安全性進行評估。這部分使用了三種測試指標，分別為 entropy、histogram、correlation。

   在 entropy 的分析上，計算了原始圖像與加密圖像間的 entropy，而 entropy 高代表著圖像被加密後的隨機性與不確定性越高。下圖可以看到在 Entropy 欄位的部分，加密後的 Entropy 數字都非常接近 12 (12 是 MAX)，顯示了這個方法加密後的隨機性是非常好的。關於 correlation 測試，若加密後的圖像與原先的圖像高度相關會容易被透過分析加密後的圖像反推原始圖像的內容，因此在 correlation 的部分加密後的圖像應該與原始圖像具有越低的

關聯度越好。而在我們的測試結果中，不論是在水平方向、垂直方向、對角線方向上的 correlation 皆趨近於 0。

	Entropy	autocorrelation		
		Horizontal	Vertical	Diagonal
圖 1(肺部 CT)	7.134	0.0026	0.0030	-0.0005
圖 2(手部 CT)	7.731	0.0009	0.0012	0.0009
圖 3(腦部 MRI)	5.222	-0.0021	0.0008	-0.0011

而 histogram 的部分，越平均的分布顯示每個像素值的出現具有相同的頻率。而下圖顯示了三種原先在 histogram 統計上具有一定差異的圖像在進行加密後的 histogram 表現都非常接近平坦的直方圖。



2. 密鑰敏感度 (Key Sensitivity)

此方法透過稍微修改密鑰並且分析輸入相同圖像加密後的些微變化來觀察密文的變化從而反推出解密的方法。並且透過 NPCR 與 UACI 衡量單一個位元改變後的平均變化程度以及每次變化的平均強度，並且在測試的圖片中不論是 NPCR 或是 UACI 都有一定的強度，顯示了其對抗差分密碼分析的強度。

而我們在分析密鑰敏感度時，分別觀察了 sbox 以及 chaotic map 的密鑰敏感度，實驗數據如下圖，在我們測試的數組資料中都顯示著他們保有一定的強度上的密鑰敏感度。

	sbox		chaotic map	
	NPCR	UACI	NPCR	UACI
圖 1(肺部 CT)	99.983	128.24	99.952	128.14
圖 2(手部 CT)	99.969	25.71	99.951	25.48
圖 3(腦部 MRI)	99.973	20.58	99.945	21.16

3. 抗選擇明文攻擊能力 (Resistance to Chosen Plaintext Attacks)

首先，此加密方法先使用了 PRNG 進行隨機加密子密鑰生成，並且用於初始化 S-box 以及 Baker map，這加強了此加密方法的抗明文攻擊能力。再者，每次加密過程都會生成一次性的 S-box，即便攻擊者破解了這個，也沒辦法透過這個方法再對其他圖像進行破解，這為抗選擇明文攻擊增添了一道防線。最後，還有兩輪的 S-box shuffling，這個方法已經被證明能夠抵禦選擇明文攻擊，也應用在這個加密法上，並且使其對於選擇明文攻擊的抵抗能力更加強大。

4. 密鑰空間分析 (Key Space Analysis)

密鑰空間在某部分可以用來衡量這個加密方法的安全性，太小的密鑰空間會使此加密方法變得不安全，不過此論文中的密鑰空間可以允許密鑰大小最大達到 19937 bits，可以說是十分的安全。

根據以上四個分析，這個加密方法在各個層面的安全性上都有被證明是有效且可靠的，不論是透過統計的 Entropy 數值，或是各項如密鑰攻擊、明文選擇攻擊，又或是分析能否以防暴力破解的密鑰空間分析，這個透過 sbox 以及 chaotic map 進行加密的 12bit 醫學影像加密方法是一個安全且可用的方法。

附錄 – 程式實作說明

1. 安裝所需的密碼學 library：

   - PyCryptodome 提供我們處理隨機數生成

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
```

2. 初始化步驟:

   - 生成隨機序列

     使用 CBC 模式和隨機初始化向量來建立 AES 加密器，使用加密器對區塊進行加密，最後將生成的隨機序列裁剪成所需的長度。

```
## Generate random sequence
def generate_random_sequence(key, sequence_length):
    # 建立加密器
    cipher = AES.new(key, AES.MODE_CBC, IV=get_random_bytes(AES.block_size))
    random_sequence = bytearray()

    while len(random_sequence) < sequence_length:
        # 使用 AES 加密器加密一個區塊
        block = cipher.encrypt(get_random_bytes(AES.block_size))
        random_sequence.extend(block)

    # 裁剪為所需的長度
    return bytes(random_sequence[:sequence_length])
```

   - 初始化 PRNG

     使用 AES 金鑰生成一個長度適中的隨機序列，再使用此隨機序列作為種子初始化 PRNG。

```
## Create PRNG
def initialize_prng(seed_length):
    # 使用 AES 金鑰產生隨機序列
    aes_key = get_random_bytes(AES.block_size)
    random_sequence = generate_random_sequence(aes_key, seed_length)

    # 使用隨機序列作為種子初始化 PRNG
    prng = np.random.default_rng(int.from_bytes(random_sequence, byteorder='big'))
```

```
    return prng
```

- 隨機序列生成 demo

```
# example
seed_length = 16

prng = initialize_prng(seed_length)
random_numbers = prng.random(size=10)    # 生成10個隨機數
print("Random numbers:", random_numbers)
```

```
Random numbers: [0.01818745 0.80179752 0.27692886 0.90565171 0.38490635 0.10750326
 0.62338828 0.04895788 0.10043544 0.37437175]
```

3. 實作動態 S-box：

- 使用 Fisher-Yates shuffle 演算法對初始 S-box 值進行洗牌

Fisher-Yates shuffle 是一種用於隨機打亂數列的演算法，其核心思想是在遍歷數列的同時，隨機選擇一個未打亂的元素並與當前元素交換。這樣可以確保每個元素都有相同的機率出現在任何位置。

```
## Fisher-Yates shuffle 演算法
def fisher_yates_shuffle(arr, rng):
    for i in range(len(arr) - 1, 0, -1):
        j = rng.integers(0, i + 1)
        arr[i], arr[j] = arr[j], arr[i]
    return arr
```

- 建構動態 S-box 的函數

根據 seed 作為隨機數種子初始化 PRNG，接下來生成範圍為 pixel_range 的初始陣列，並使用 Fisher-Yates shuffle 演算法進行打亂，最後得到動態 S-box。

```
def construct_sbox(seed, pixel_range):
    rng = np.random.default_rng(seed) # 使用 seed 整數作為隨機數種子創建 PRNG
    sbox = np.arange(pixel_range, dtype=np.uint16)
    sbox = fisher_yates_shuffle(sbox, rng)
    return sbox
```

4. 實作混沌映射和遮罩:

- 定義 Baker 混沌映射

```
## 定義 Baker 映射
def baker_map(x, y, a, b):
  x_new = a * (x + y) % 1
```

```
    y_new = b * (x - y) % 1
    return x_new, y_new
```

- 利用 Baker 混沌映射建構三維的混沌遮罩

使用 seed 初始化 PRNG，再使用 Baker 混沌映射更新參數值，並生成隨機的遮罩值給影像中的每個像素。

```
## 生成 3D 混沌遮罩
def generate_chaotic_mask_3d(seed, image_size, a=0.921064, b=0.040442):
    chaotic_mask = np.zeros(image_size, dtype=np.uint16)  # 使用 16 位元表示
    rng = np.random.default_rng(int.from_bytes(seed, byteorder='big'))
    x = rng.random()
    y = rng.random()
    for i in range(image_size[0]):
        for j in range(image_size[1]):
            for k in range(image_size[2]):
                x, y = baker_map(x, y, a, b)
                chaotic_mask[i, j, k] = rng.integers(0, 4096)
    return chaotic_mask
```

5. 圖像加解密過程:

- 加密函數

先初始化影像的大小與類型，使其與原始影像相同。接著針對每個像素使用混沌遮罩與動態 S-box 進行像素替換，最後將加密後的像素值填入影像的對應位置。

```
# 加密影像
def encrypt_image_3d(plain_image, sbox, chaotic_mask):
    encrypted_image = np.zeros_like(plain_image, dtype=np.uint16)

    for i in range(plain_image.shape[0]):
        for j in range(plain_image.shape[1]):
            for k in range(plain_image.shape[2]):
                I_i = plain_image[i, j, k] # 取得原始像素值
                M_i = chaotic_mask[i, j, k] # 取得混沌遮罩值
                M_i_prime = int(bin(M_i)[2:][::-1], 2) % 256  # 翻轉二進位表示

                # 透過動態 S-box 進行像素替換
                encrypted_pixel = sbox[M_i_prime ^ sbox[M_i ^ sbox[I_i]]]
```

```
            encrypted_image[i, j, k] = encrypted_pixel
    return encrypted_image
```

- 解密函數

先初始化影像的大小與類型，使其與原始影像相同，接下來計算用於解密過程的反向 S-box。針對每個像素提取混沌遮罩值，再按照解密公式進行解密，最後將解密後的像素填入影像的對應位置。

```
# 解密影像
def decrypt_image_3d(encrypted_image, sbox, chaotic_mask):

    # 初始化解密影像 (使用 uint8 表示像素值)
    decrypted_image = np.zeros_like(encrypted_image, dtype=np.uint8)

    inv_sbox = np.zeros_like(sbox) # 計算 S-box 的反向查找表
    for i in range(len(sbox)):
        inv_sbox[sbox[i]] = i

    for i in range(encrypted_image.shape[0]):
        for j in range(encrypted_image.shape[1]):
            for k in range(encrypted_image.shape[2]):
                C_i = encrypted_image[i, j, k]

                M_i = chaotic_mask[i, j, k] # 取得混沌遮罩值
                M_i_prime = int(bin(M_i)[2:][::-1], 2) % 256 # 翻轉二進位表示

                S_C_i = inv_sbox[C_i] # 取得密文像素值的反向查找表值
                inner_value = M_i_prime ^ S_C_i
                S_inner_value = inv_sbox[inner_value]
                middle_value = M_i ^ S_inner_value
                I_i = inv_sbox[middle_value]

                # 將解密後的像素值填入解密影像
                decrypted_image[i, j, k] = I_i
    return decrypted_image
```

6. 實作結果展示：

- 初始化與圖像加密

先調整輸入圖像的大小，再建構三維混沌遮罩，搭配前面建立好的動態 S-box 一起進行圖像加密。

```
plain_image = Image.open(PLAIN_IMAGE_PATH)
image_array, sbox, chaotic_mask_3d, sbox_2, chaotic_mask_3d_2 =
get_image_data(plain_image)
print("Image data loaded successfully.")
# 加密圖像
encrypted_image = encrypt_image_3d(image_array, sbox, chaotic_mask_3d)
print("Image encrypted successfully.")
```

```
def get_image_data(plain_image):
    image_size = (plain_image.size[1], plain_image.size[0], 3)
    image_array = np.array(plain_image.resize((image_size[1], image_size[0])))

    # hint : 若圖像為灰度，則轉為三通道
    if len(image_array.shape) == 2:
        image_array = np.stack((image_array,)*3, axis=-1)

    prng = initialize_prng(SEED_LEN)  # PRNG 初始化
    sbox_seed = prng.integers(0, 2**32)
    sbox = construct_sbox(sbox_seed, PIXEL_RANGE) # 建立 S-Box
    modified_seed = sbox_seed
    modified_seed ^= 1
    sbox_2 = construct_sbox(modified_seed, PIXEL_RANGE) # 建立 S-Box2

    chaotic_seed = prng.integers(0, 2**32)
    chaotic_mask_3d = generate_chaotic_mask_3d(chaotic_seed, image_size)
    modified_seed = chaotic_seed
    modified_seed ^= 1
    chaotic_mask_3d_2 = generate_chaotic_mask_3d(modified_seed,
 image_size)

    return image_array, sbox, chaotic_mask_3d, sbox_2, chaotic_mask_3d_2
```

- 影像資料歸一化

在使用 12 位元混沌遮罩進行加密後，產生的影像資料可能超出了 imshow 函數的預期範圍。

為了在不改變 12 位元混沌遮罩的前提下顯示加密後的影像，我們將影像資料歸一化，重新縮放到 imshow 可接受的範圍 [0, 1]

```
# hint : 將加密影像資料歸一化到 [0, 1]
 encrypted_image_display = (encrypted_image / encrypted_image.max())
```

- 圖像解密與成果展示

```
# 解密圖像
decrypted_image = decrypt_image_3d(encrypted_image, sbox, chaotic_mask_3d)
print("Image decrypted successfully.")

# 顯示結果
display_image(image_array, encrypted_image_display, decrypted_image)
```

```
def display_image(plain_image, encrypted_image_display, decrypted_image):
    fig, axes = plt.subplots(1, 3, figsize=(15, 5))  # Create a figure and subplots

    images = [plain_image, encrypted_image_display, decrypted_image]
    titles = ["Plain Image", "Encrypted Image", "Decrypted Image"]

    for ax, img, title in zip(axes, images, titles):
        ax.imshow(img, cmap='gray')
        ax.set_title(title)
        ax.axis('off')
    print("Displaying results...")
    plt.show()
```

- 輸出