Faculty of Engineering & Technology

Electrical & Computer Engineering Department

APPLIED CRYPTOGRAPHY – ENCS4320

Homework1

Prepared by:

Name: Mohammad Abu Shams **ID**: 1200549

Instructor: Dr. Ahmed Shawahna

Section: 1

Date: 25-12-2024

BIRZEIT

Table of Contents

Question1

Using your cryptanalysis skills, find the *plaintext* (and the *key*) that corresponds to the ciphertext ***WLIMWXLIWSYPSJQCWSYP***, given that the ***shift cipher (ROT-k)*** was used.

Ciphertext: WLIMWXLIWSYPSJQCWSYP

Using the ROT-k shift cipher, I tried shifts from 1 to 4. The correct message appeared with shift 4

Decryption: $Mi = Ci - K \pmod{26}$

Key	Plaintext
1	vkhlvwkhvrxoripbvrxo
2	ujgkuvjguqwnqhoauqwn
3	tifjtuiftpvmpgnztpvm
4	sheisthesoulofmysoul

The plaintext is: (sheisthesoulofmysoul) (she is the soul of my soul)

The key (k) is: 4

Question2

Assume an attacker knows that a user's password is either "***wxyz***" or "***bddf***". Say the user encrypts his password using:

a) The substitution cipher,
b) The Vigenère cipher using period 2, or
c) The Vigenère cipher using period 3

and the attacker sees the resulting ciphertext. Show how the attacker can determine the user's password, or explain why this is not possible.

a) The substitution cipher

In a substitution cipher, each letter in the message is replaced by a fixed letter based on a key. This means that the same letter, like 'd', will always turn into the same letter in the encrypted message.

To figure out which password was encrypted, we can check the second and third letters of the ciphertext:

- If the second and third letters are the same, the ciphertext is from the second password.
- If they are different, the ciphertext is from the first password.

b) The Vigenère cipher using period 2,

When the period is 2, we can't tell which password was encrypted. This happens because:

- The same shift is used for the first and third positions, and for the second and fourth positions.
- The difference between the first and third letters (and second and fourth letters) is the same in both passwords.

c) The Vigenère cipher using period 3

When the period is 3, we can figure out which password was encrypted. This is because:

- The same shift is used for the first and fourth letters.
- The difference between the first and fourth letters in the first password is not the same as in the second password.

For the ciphertext **C: C0C1C2C3**:

- If **C3 - C0 (mod 26) = 3** (like z[25] - w[22]), the ciphertext is from the first password.
- If **C3 - C0 (mod 26) = 4** (like f[5] - b[1]), the ciphertext is from the second password.

Question3

Suppose we have a computer with a 4.2 GHz 16-core processor that executes 4.2×10^9 cycles per second per core. Considering that it can test a key per CPU cycle:

a) What is the expected time (in years) to find a key by the brute-force attack if the key size is **56** bits?
b) What is the expected time (in years) to find a key by the brute-force attack if the key size is **128** bits?

Number of keys can be tested per second = Number of cycles per second per core*Number of keys can be tested per CPU cycle*Number of cores in the PC

Number of keys can be tested per second = $(4.2*10^9) * 1 * 16 = 6.72 * 10^{10}$

a) What is the expected time (in years) to find a key by the brute-force attack if the key size is 56 bits?

Number of keys = 2^{56}

Expected time (in Seconds) = Number of keys / Number of keys can be tested per second

= 2^{56} /($6.72 * 10^{10}$)

= 1072285.626 seconds = 17870.9771 minutes = 297.8496183 hours = 12.41040076 days
= **0.033977825 years**.

b) What is the expected time (in years) to find a key by the brute-force attack if the keysize is 128 bits?

Number of keys = 2^{128}

Expected time (in Seconds) = Number of keys / Number of keys can be tested per second

= 2^{128}/($6.72 * 10^{10}$)

= $5.063725698 * 10^{27}$ seconds= $8.43954283* 10^{25}$ minutes = $1.406590472 * 10^{24}$ hours

= $5.860793632 * 10^{22}$ days

= **$1.604597846 * 10^{20}$ years**.

Question4

Alice is using the one-time pad and notices that when her key is all-zeroes $K = 0^n$, then $\text{Enc}(K, M) = M$ and her message is sent in the clear! To avoid this problem, she decides to modify the scheme to exclude the all-zeroes key. That is, the key is now chosen uniformly from $\{0,1\}^n \setminus \{0^n\}$, the set of all n-bit strings except 0^n. In this way, she guarantees that her plaintext

is never sent in the clear. Is this variant still one-time perfectly secure? Justify your answer.

The key space |K| is 2^{n-1} and the message space |M| is 2^n, meaning there are more possible messages than keys. Since there are fewer keys than messages, the system is not perfectly secure.

Proof:

C1 ⊕ C2 = (M1 ⊕ Key) ⊕ (M2 ⊕ Key)

=Key ⊕ Key ⊕ M1 ⊕ M2

= 0 ⊕ M1 ⊕ M2

=C1 ⊕ C2 = M1 ⊕ M2

An attacker can find out the message.

In a perfectly secure system, the chance of guessing a message M correctly should stay the same before and after seeing the encrypted message C:

- **P(M = m | C = c) = P(M = m)**.
- P(M=m) is the probability of guessing a message correctly is $1/(2^n)$.
- P(M=m | C=c)=0 , after seeing the ciphertext, the probability becomes 0.

This difference between prior and posterior information shows that the system is not perfectly secure.

Question5

Answer each of the following without using a calculator.

a) $3 - 11 \pmod 9 =$

b) $15 \times 29 \pmod{13} =$

c) $-12 / 35 \pmod{19} =$

d) Are 172 and 68 co-prime numbers?

a) 3 −11 (mod 9)

= −8 (mod 9) ≡ 1 (9-8=1)

b) 15 × 29 (mod 13)

(15 (mod 13) ×29 (mod 13)) mod 13 = 2×3 (mod 13) = 6 mod 13 =6

c) −12 / 35 (mod 19)

= (−12 (mod 19) $\times 35^{-1}$ (mod 19)) mod 19

−12 (mod 19) = 7

i	qi-1	ri	si	ti
0		35	1	0
1		19	0	1
2	1	16	1	-1
3	1	3	-1	2
4	5	1	6	-11
5		0		

6*35 +-11*19=1

6*35 mod 19 = 1 mod 19

So 35^{-1} (mod 19) = 6

So −12 / 35 (mod 19) = 7×6 (mod 19)

=42 mod 19 =4

d) Are 172 and 68 co-prime numbers?

172 and 68 are co-prime if their only common factor is 1.

gcd(r0,r1)=gcd(r1,ro mod r1)

gcd(172,68)=gcd(68,36)= gcd (36,32)= gcd (32,4)= gcd (4,0)=4 ≠1

So 172 and 68 are not relatively prime numbers.

Question6

The following questions concern multiple encryptions of single-character ASCII plaintexts with the one-time pad using the same 8-bit key. You may assume that the plaintexts are either (upper-case or lower-case) English letters or space character. Note that the ASCII code for the space character is 20 (hex) = 0010 0000 (binary), the ASCII code for 'A' is 41 (hex) = 0100 0001 (binary), and the ASCII code for 'a' is 61 (hex) = 0110 0001 (binary), as it is clear from the table below.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	space	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

a) Say you see the ciphertexts 3D (hex) and 44 (hex). What can you deduce about the plaintext characters these correspond to?

- C1 = 3D (hex) = 0011 1101 (binary)
- C2 = 44 (hex) = 0100 0100 (binary)
- Since the same key is used, C1 ⊕ C2 = M1 ⊕ M2.
- C1 ⊕ C2 = 0111 1001.

Looking at the result, it can be concluded from the bits with orange color that one message is a capital letter and the other is a space.

- A space in binary is 0010 0000.

To find the other message, space ⊕ result:

- 0010 0000 ⊕ 0111 1001 = 0101 1001.
- 0101 1001 is 59 in hex, which represents the letter Y.

b) Say you see the three ciphertexts FF (hex), B5 (hex), and C7 (hex). What can you deduce about the plaintext characters these correspond to?

- C1 = FF(hex) = 1111 1111 (binary)
- C2 = B5 (hex) = 1011 0101 (binary)
- C3 = C7 (hex) =1100 0111 (binary)

XOR operations:

- C1 ⊕ C2 = M1 ⊕ M2 = 0100 1010
  - This means one message is a small letter, and the other is a space.
- C1 ⊕ C3 = M1 ⊕ M3 = 0011 1000
  - This shows one message is a small letter, and the other is a capital letter.
- C2 ⊕ C3 = M2 ⊕ M3 = 0111 0010
  - This indicates one message is a capital letter, and the other is a space.

We conclude that M2 is a space (binary: 0010 0000), we can find M1 and M3:

- M1 is a small letter.
- M3 is a capital letter.

Finding the exact values:

- M1:
  Space ⊕ (C1 ⊕ C2) = 0010 0000 ⊕ 0100 1010 = 0110 1010 = 6A (hex) = j.
- M3:
  Space ⊕ (C2 ⊕ C3) =0010 0000 ⊕ 0111 0010 = 0101 0010 = 52 (hex) = R.

So, M1 = j and M3 = R.

Question7

Suppose that, after a particular step of ***A5/1 stream cipher***, the values in the registers are:

$$X = (x_0, x_1, \ldots, x_{18}) = (1010101010101010110)$$
$$Y = (y_0, y_1, \ldots, y_{21}) = (1100110001101100010011)$$
$$Z = (z_0, z_1, \ldots, z_{22}) = (11100101110000011000011)$$

a) List the next 4 keystream bits.

b) Give the contents of X, Y, and Z after the generation of each of these 4 bits.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**
x	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1	0				
y	1	1	0	0	1	1	0	0	0	1	1	0	1	1	0	0	0	1	0	0	1	1	
z	1	1	1	0	0	1	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0	1	1

Step1:

Maj (x8,y10,z10) = maj (1,1,0)=1

So X step, y step, z does not step.

For X, since X8 = maj(1, 1, 0), X0 = X13 ⊕ X16 ⊕ X17 ⊕ X18 = 0 ⊕ 1 ⊕ 1 ⊕ 0 = 0

For Y, since Y10 = maj(1, 1, 0), Y0 = Y20 ⊕ Y21 = 1 ⊕ 1 = 0

For Z, since Z10 ≠maj(1, 1, 0), nothing happens

X, Y, Z After generation step1:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**
x	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1				
y	0	1	1	0	0	1	1	0	0	0	1	1	0	1	1	0	0	0	1	0	0	1	
z	1	1	1	0	0	1	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0	1	1

First stream bit = x18 ⊕ y21 ⊕ z22= 1 ⊕ 1 ⊕ 1 = 1

****

Step2:

Maj (x8,y10,z10) = maj (0,1,0)=0

So X step, y does not step, z step

For X, since X8 = maj(0, 1, 0), X0 = X13 ⊕ X16 ⊕ X17 ⊕ X18 = 1 ⊕ 0 ⊕ 1 ⊕ 1 = 1

For Y, since Y10 ≠ maj(0, 1, 0), nothing happens.

For Z, since Z10 =maj(0, 1, 0), Z0=Z7 ⊕ Z20 ⊕ Z21 ⊕ Z22=1 ⊕ 0 ⊕ 1 ⊕ 1=1

X, Y, Z After generation step2:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
x	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1				
y	0	1	1	0	0	1	1	0	0	0	1	1	0	1	1	0	0	0	1	0	0	1	
z	1	1	1	1	0	0	1	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0	1

Second stream bit = x18 ⊕ y21 ⊕ z22=1 ⊕ 1 ⊕ 1 = 1

****

Step3:

Maj (x8,y10,z10) = maj (1,1,1)=1

So X step, y step, z step

For X, since X8 = maj (1,1,1),X0 = X13 ⊕ X16 ⊕ X17 ⊕ X18 = 0 ⊕ 1 ⊕ 0 ⊕ 1 = 0

For Y, since Y10 = maj (1, 1, 1), Y0 = Y20 ⊕ Y21 = 0 ⊕ 1 = 1

For Z, since Z10 =maj(1, 1, 1), Z0=Z7 ⊕ Z20 ⊕ Z21 ⊕ Z22=0 ⊕ 0 ⊕ 0 ⊕ 1=1

X, Y, Z After generation step3:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
x	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0				
y	1	0	1	1	0	0	1	1	0	0	0	1	1	0	1	1	0	0	0	1	0	0	
z	1	1	1	1	1	0	0	1	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0

Third stream bit = x18 ⊕ y21 ⊕ z22=0 ⊕ 0 ⊕ 0 = 0

Step4:

Maj (x8,y10,z10) = maj (0,0,1)=0

So X step, y step, z does not step.

For X, since X8 = maj(1, 1, 0), X0 = X13 ⊕ X16 ⊕ X17 ⊕ X18 = 1 ⊕ 0 ⊕ 1 ⊕ 0 = 0

For Y, since Y10 = maj(1, 1, 0), Y0 = Y20 ⊕ Y21 = 0 ⊕ 0 = 0

For Z, since Z10 ≠maj(1, 1, 0), nothing happens

X, Y, Z After generation step4:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**
x	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1				
y	0	1	0	1	1	0	0	1	1	0	0	0	1	1	0	1	1	0	0	0	1	0	
z	1	1	1	1	1	0	0	1	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0

Fourth stream bit = x18 ⊕ y21 ⊕ z22=1 ⊕ 0 ⊕ 0 = 1

So the next 4 keystream bits = *k*0*k*1*k*2*k*3 = 1101.

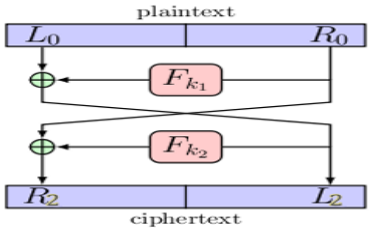
****

Question8

Consider a new ***block cipher***, **DES2**, that consists of only *two rounds* of the **DES** block cipher. **DES2** has the *same block and key size* as DES. For this question, you should consider the DES F function as a black box that takes two inputs, a 32-bit data segment, and a 48-bit round key, and produces a 32-bit output. Using the chosen-plaintext attack (CPA) without any restrictions on the number of oracle calls.

a) Give an algorithm to recover the 48-bit round keys for round 1 (k_1) and round 2 (k_2). Your algorithm should have fewer operations than the exhaustive key search for **DES2**.

b) Can your algorithm be converted into a distinguishing attack against **DES2**, i.e., an attack that distinguishes **DES2** from a random permutation?



a) Give an algorithm to recover the 48-bit round keys for round 1 (*k*1) and round 2 (*k*2). Your algorithm should have fewer operations than the exhaustive key search for DES2.

The round keys for round 1 (**k1**) and round 2 (**k2**) are used to define the functions **Fk1** and **Fk2**. These functions take a 32-bit input and produce a 32-bit output. We need to calculate the lookup tables for these functions.

Step 1: Calculate Fk1

- The function **Fk1** is given by:
  L2 = L0 ⊕ Fk1(R0)
  - If **L0 = 0 (32 zeros)**, then **Fk1(R0) = L2**.
- To create the lookup table for **Fk1**:
  - Set **L0 = 0**.
  - For every possible value of **R0** (from 0 to $2^{32} - 1$):
    - Compute **Fk1(R0) = L2**.

Step 2: Calculate Fk2

- The function **Fk2** is given by:
  R2 = R0 ⊕ Fk2(L0 ⊕ Fk1(R0))
  - If **R0 = 0 (32 zeros)**, then **Fk1(R0)** is a constant value **C** (calculated in Step 1).
  - So, **Fk2(L0 ⊕ C) = R2**.
- To create the lookup table for **Fk2**:
  - Set **R0 = 0** and calculate **C = Fk1(R0)**.
  - For every possible value of **L0** (from 0 to $2^{32} - 1$):
    - Compute **Fk2(L0 ⊕ C) = R2**.

Number of Operations

- Calculating **Fk1** requires 2^{32} operations.
- Calculating **Fk2** also requires 2^{32} operations.
- Total operations: 2^{33}, which is much smaller than the 2^{56} operations needed for a full DES2 key search.

b) Can your algorithm be converted into a distinguishing attack against DES2, i.e., an attack that distinguishes DES2 from a random permutation?

To identify if a cipher is **DES2** or a random permutation, we can perform the following attack:

Step 1: Send the first pair

- Choose random values **X1** and **Y1** (each 32 bits).
- Send **(X1, Y1)** to the oracle and receive the output **(A1, B1)**.
- If the cipher is **DES2**, the output will be:
  - **A1 = X1 ⊕ Fk1(Y1)**
  - **B1 = Y1 ⊕ Fk2(X1 ⊕ Fk1(Y1))**

Step 2: Send the second pair

- Choose a new random value **X2** (32 bits) while keeping the same **Y1**.
- Send **(X2, Y1)** to the oracle and receive the output **(A2, B2)**.
- If the cipher is **DES2**, the output will be:
  - **A2 = X2 ⊕ Fk1(Y1)**
  - **B2 = Y1 ⊕ Fk2(X2 ⊕ Fk1(Y1))**

Step 3: Verify the results

- Calculate **A1 ⊕ A2**:
  - **A1 ⊕ A2 = (X1 ⊕ Fk1(Y1)) ⊕ (X2 ⊕ Fk1(Y1))**
  - This simplifies to **A1 ⊕ A2 = X1 ⊕ X2** because the **Fk1(Y1)** terms cancel out.
- If **A1 ⊕ A2 = X1 ⊕ X2**, then it is likely the cipher is **DES2**.
- For a random permutation, the probability of this happening by chance is about 2^{-32} making it extremely unlikely.

This test helps confirm whether the cipher being used is **DES2**.

Question9

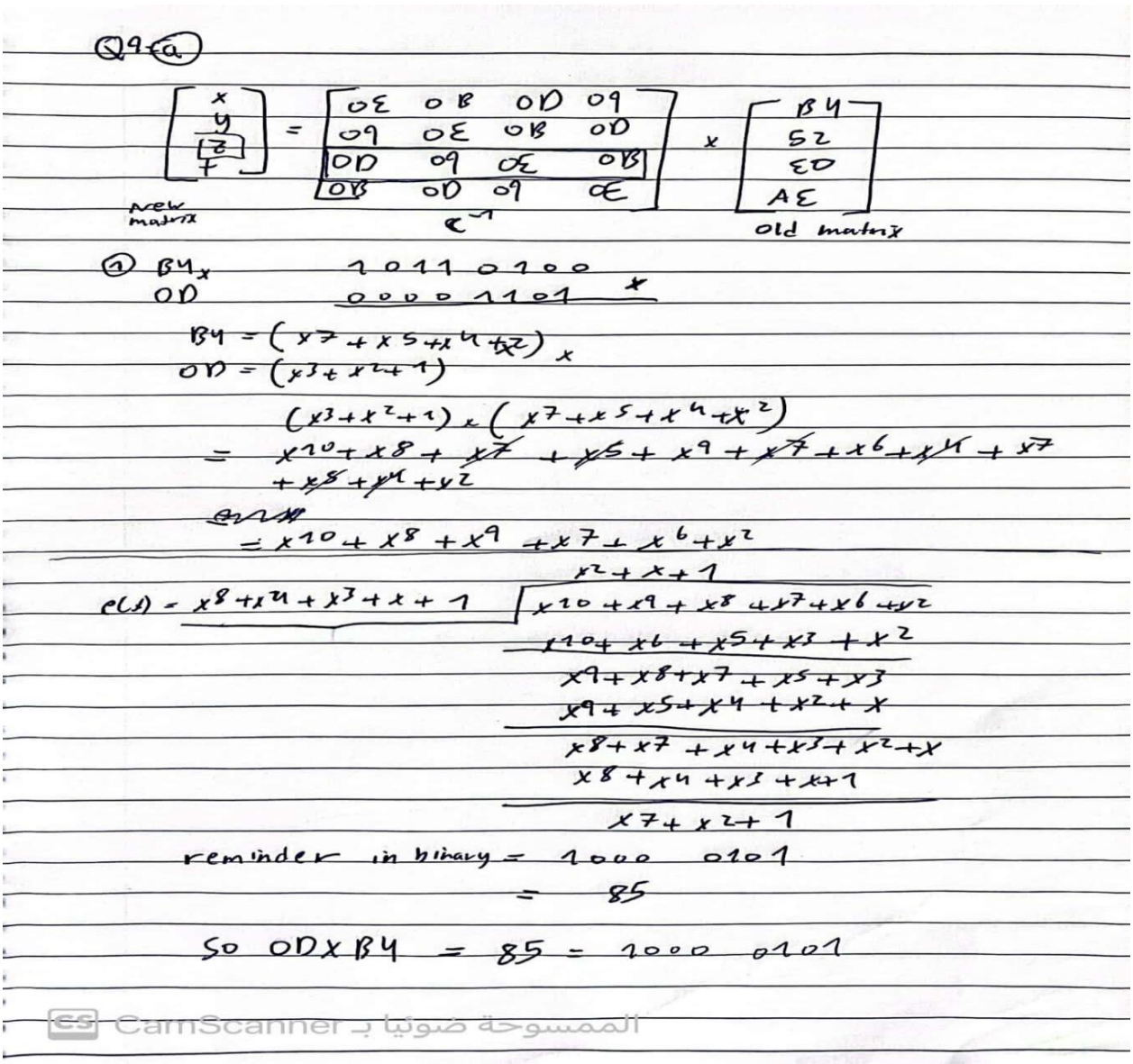
This problem deals with the ***AES-128 block cipher***.

a) Assume that the first column of the input to the InvMixColumn step is $S_{i,0}$ = (B4, 52, E0, AE)$_{16}$, find the 3rd element of the corresponding column of the output state of the InvMixColumn step.

b) Given the input $S_{2,1}$ = (7A)$_{16}$, find InvSubByte($S_{2,1}$).

c) Assume that round key 6 ($\boldsymbol{k_6}$) is (98 0F 71 AF 15 C9 47 D9 0C B7 E8 59 D6 7F 67 AD)$_{16}$, find the round keys for round 5 ($\boldsymbol{k_5}$) and round 7 ($\boldsymbol{k_7}$).

a) Assume that the first column of the input to the InvMixColumn step is Si,0 = (B4, 52, E0, AE)16, find the 3rd element of the corresponding column of the output state of the InvMixColumn step.



② 52 0101 0010

×09 0000 1001 ×

$52 = x^6 + x^4 + x$

$09 = x^3 + 1$

$(52 \times 09) = (x^6 + x^4 + x)(x^3 + 1)$

$= x^9 + x^6 + x^7 + x^4 + x^4 + x$

$= x^9 + x^7 + x^6 + x$

$p(x) = x^8 + x^4 + x^3 + x + 1$

$$\begin{array}{r|l} x^8 + x^4 + x^3 + x + 1 & x \\ \hline & x^9 + x^7 + x^6 + x \\ & x^9 + x^5 + x^4 + x^2 + x \\ \hline & x^7 + x^6 + x^5 + x^4 + x^2 \end{array}$$

reminder in binary = 1111 0100

= (F4)

so $(52 \times 09) = F4 = 1111\ 0100$

(3) 30 03

1110 0000

0000 1110

$(5x^3+9x^2+7x) = 03$

$(x^3+x^2+x) = 30$

$(x^3+x^2+x) \times (5x^3+9x^2+7x) = 30 \times 03$

$= x^{10}+x^9+x^8+x^9+x^8+x^7+x^8+x^7+x^6$

$= x^{10}+x^8+x^6$

$x^8+x^4+x^3+x+1 \;\big|\; x^{10}+x^8+x^6$

$x^{10}+x^6+x^5+x^2+x^3$

$x^8+x^5+x^3+x^2$

$x^8+x^4+x^3+x+1$

$x^5+x^4+x^2+x+1$

x^2+1

remider = 0011 0111

$(30 \times 03) = (37)$

④ AE 1010 1110

×OB 0000 1011

$AE = (x^7+x^5+x^3+x^2+x)$

$OB = (x^3+x+1)$

$AE \times OB = (x^3+x+1)(x^7+x^5+x^3+x^2+x)$

$= x^{10}+x^8+x^6+x^5+x^4+x^8+x^6+x^4+x^3+x^2+x^7+x^5+x^3+x^2+x$

$= x^{10}+x^7+x$

$$\begin{array}{r|l} & x^2 \\ \hline x^8+x^4+x^3+x+1 & x^{10}+x^7+x \\ & x^{10}+x^6+x^5+x^3+x^2 \\ \hline & x^7+x^6+x^5+x^3+x^2+x \end{array}$$

remainder = 1110 1110

= EE

final result = 85 ⊕ E4 ⊕ 57 ⊕ EE

1000 0101 ⊕

1111 0100

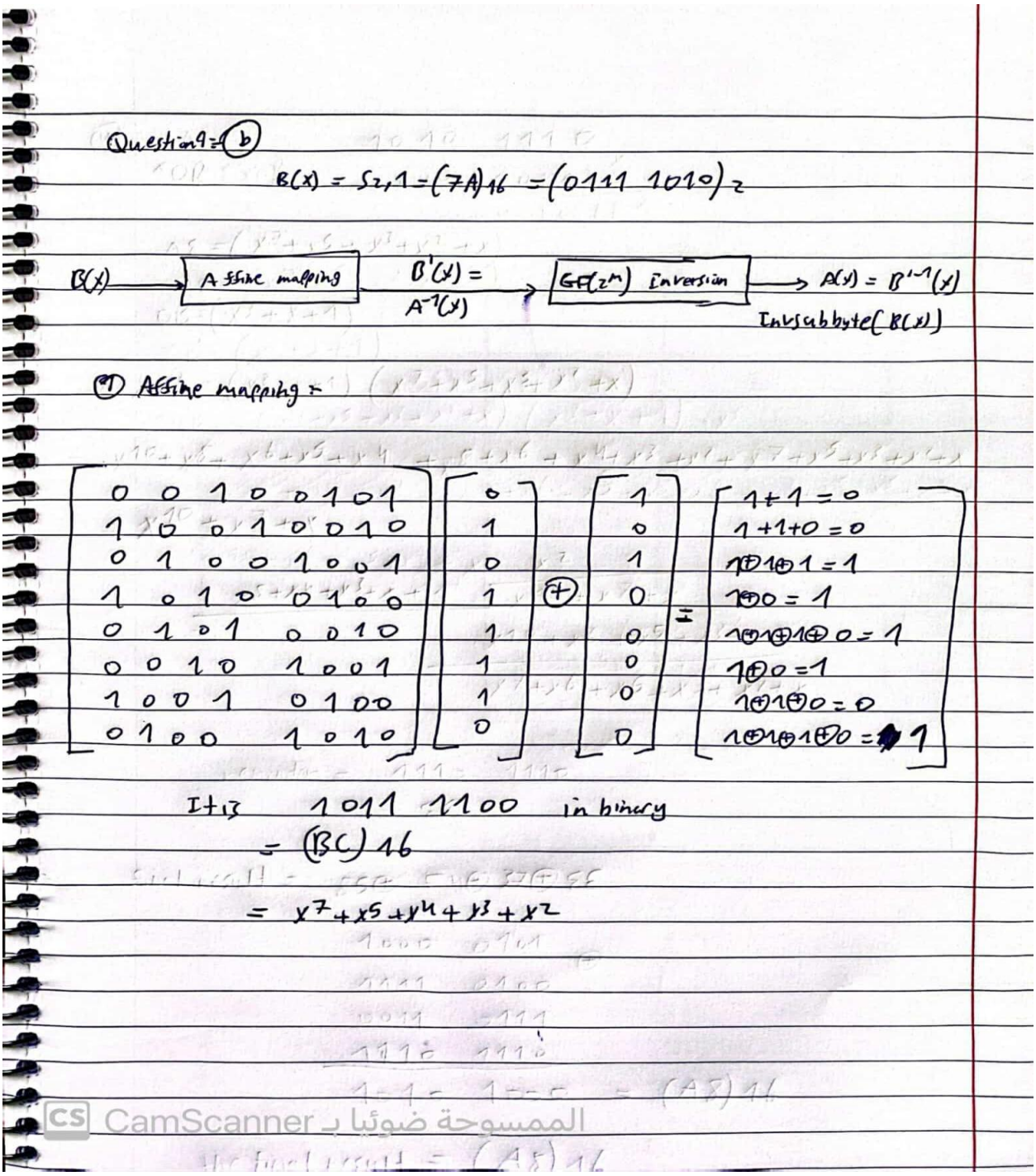
0011 0111

1110 1110

1010 1000 = (A8)16

the final result = (A8)16

b) Given the input S2,1 = (7A)16, find InvSubByte(S2,1).



④ GF(2^n) inversion :

$$(x^7+x^5+x^4+x^3+x^2)^{-1} \pmod{x^8+x^4+x^3+x+1}$$

i	q_{i-1}	r_i	s_i	t_i
0		$x^8+x^4+x^3+x+1$	1	0
1		$x^7+x^5+x^4+x^3+x^2$	0	1
2	x	x^6+x^5+x+1	1	x
3	$x+1$	x^4+x^3+1	$x+1$	x^2+x+1
4	x^2	x^2+x+1	x^3+x^2+1	$x^4+x^3+x^2+x$
5	x^2+1	x	$x^5+x^4+x^3+x$	x^6+x^5+1
6	$x+1$	1	x^6+x+1	$x^7+x^5+x^4+x^3+x^2+1$
7		0		

① $x^7+x^5+x^4+x^3+x^2 \overline{)\, x^8+x^4+x^3+x+1}$ (quotient x)

$x^8+x^6+x^5+x^4+x^3$

x^6+x^5+x+1

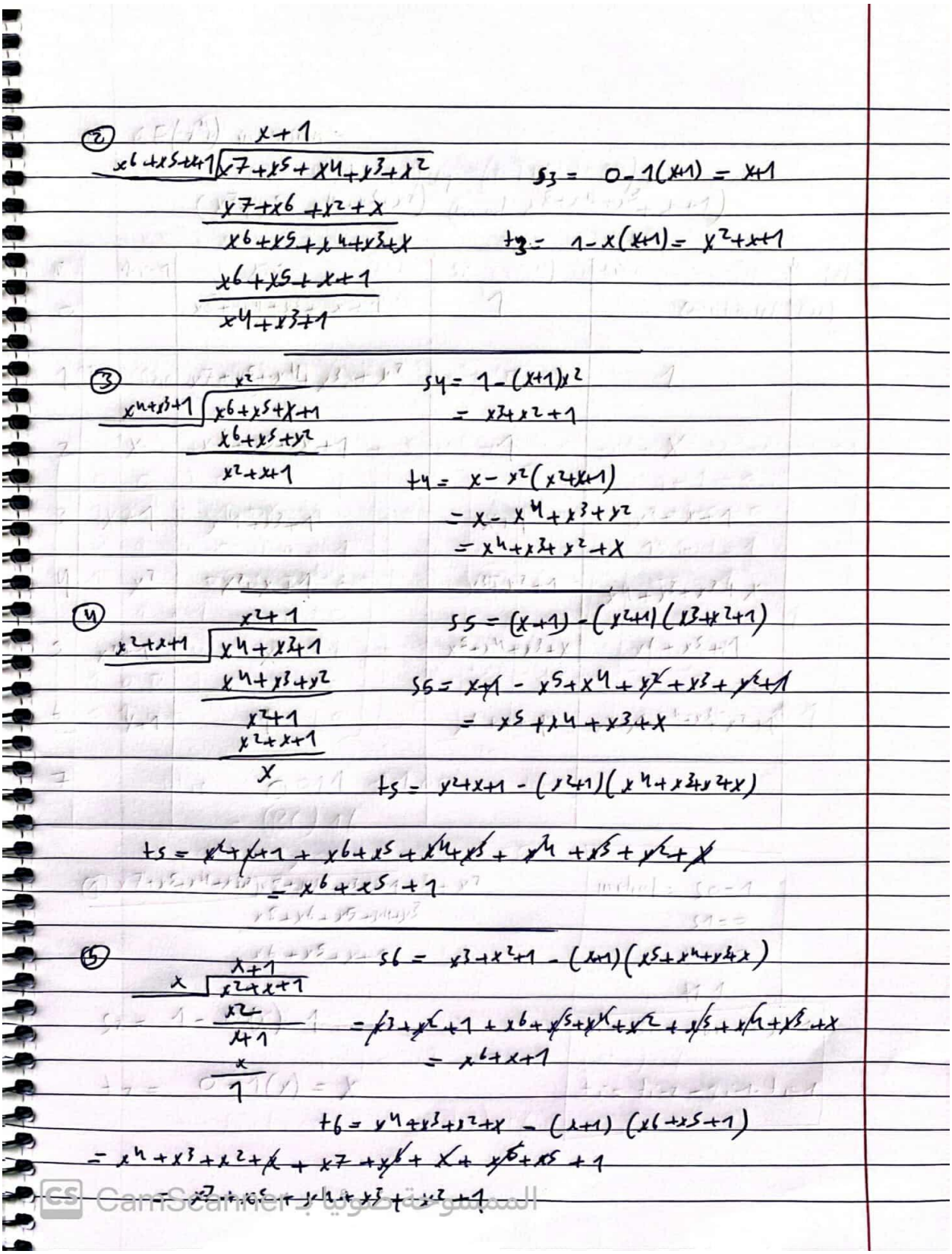
$s_2 = 1 - 0(x) = 1$

$t_2 = 0 - 1(x) = x$

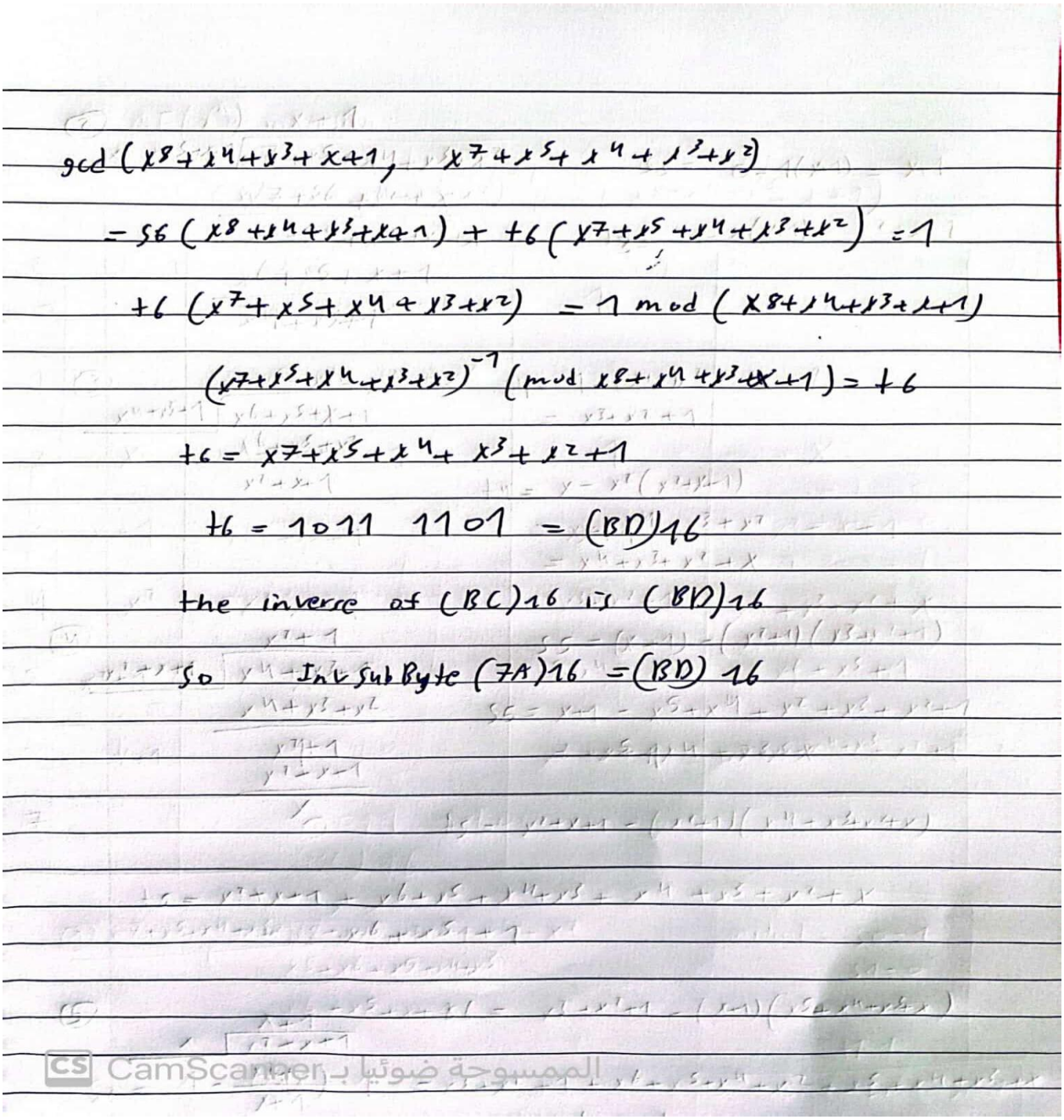
initial : $s_0=1$, $s_1=0$, $t_0=0$, $t_1=1$

$s_i = s_{i-2} - q_{i-1} \cdot s_{i-1}$

$t_i = t_{i-2} - q_{i-1} \cdot t_{i-1}$

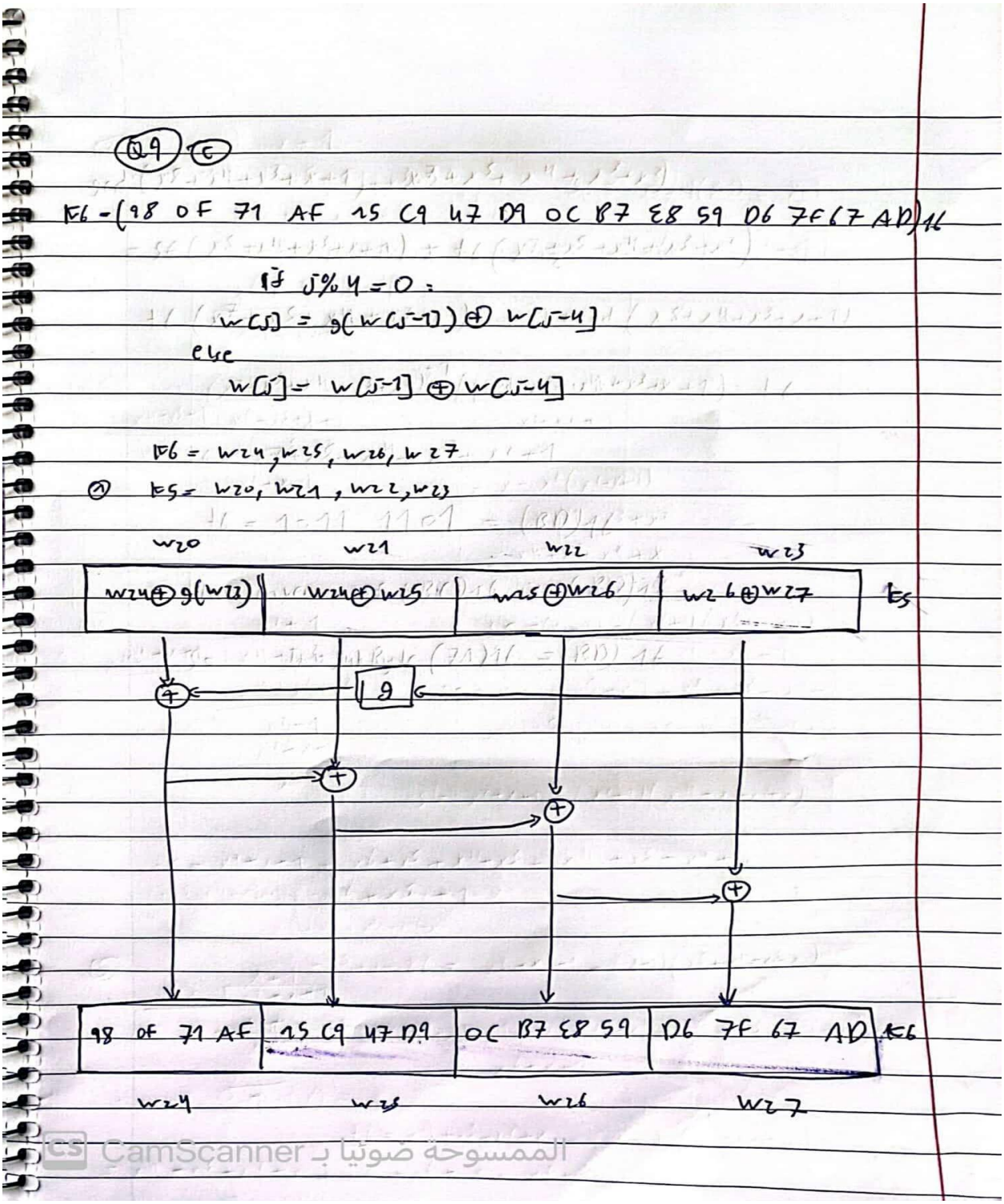


$= x^7 + x^5 + x^4 + x^3 + x^2 + 1$



c) Assume that round key 6 ($k6$) is (98 0F 71 AF 15 C9 47 D9 0C B7 E8 59 D6 7F 67 AD)16, find the round keys for round 5 ($k5$) and round 7 ($k7$).

For k5



First find $w[23] = w[26] \oplus w[27]$

0C B7 E8 59
D6 7F 67 AD ⊕

0000 1100 1011 0111 1110 1000 0101 1001
1101 0110 0111 1111 0110 0111 1010 1101 ⊕

1101 1010 1100 1000 1000 1111 1111 0100

$w[23] = (DA\ C8\ 8F\ F4)$

Find $w[22] = w[25] \oplus w[26]$

15 C9 47 D9
0C B7 E8 59 ⊕

0001 0101 1100 1001 0100 0111 1101 1001
0000 1100 1011 0111 1110 1000 0101 1001 ⊕

0001 1001 0111 1110 1010 1111 1000 0000

$w[22] = (19\ 7E\ AF\ 80)$

$w[21] = w[24] \oplus w[25]$

98 0F 71 AF
15 C9 47 D9 ⊕

1001 1000 0000 1111 0111 0001 1010 1111
0001 0101 1100 1001 0100 0111 1101 1001 ⊕

1000 1101 1100 0110 0011 0110 0111 0110

$w[21] = (8D\ C6\ 36\ 76)$

$w[20] = w[24] \oplus g(w[23])$

$w[23] = (DA\ C8\ 8F\ F4)$

RotWord(w[23]) = C8 8F F4 DA = Y6

then SubWord(Y6) = C8 73 BF 57 = Y6

Rcon(6) = X5 = 0010 0000

Rcon(6) = 20 00 00 00

Y6 ⊕ Rcon(6) = C8 73 BF 57 = g(w[23])

w[20] = 98 0F 71 AF ⊕
C8 73 BF 57

1001 1000 0000 1111 0111 0001 1010 1111
1100 1000 0111 0011 1011 1111 0101 0111 ⊕

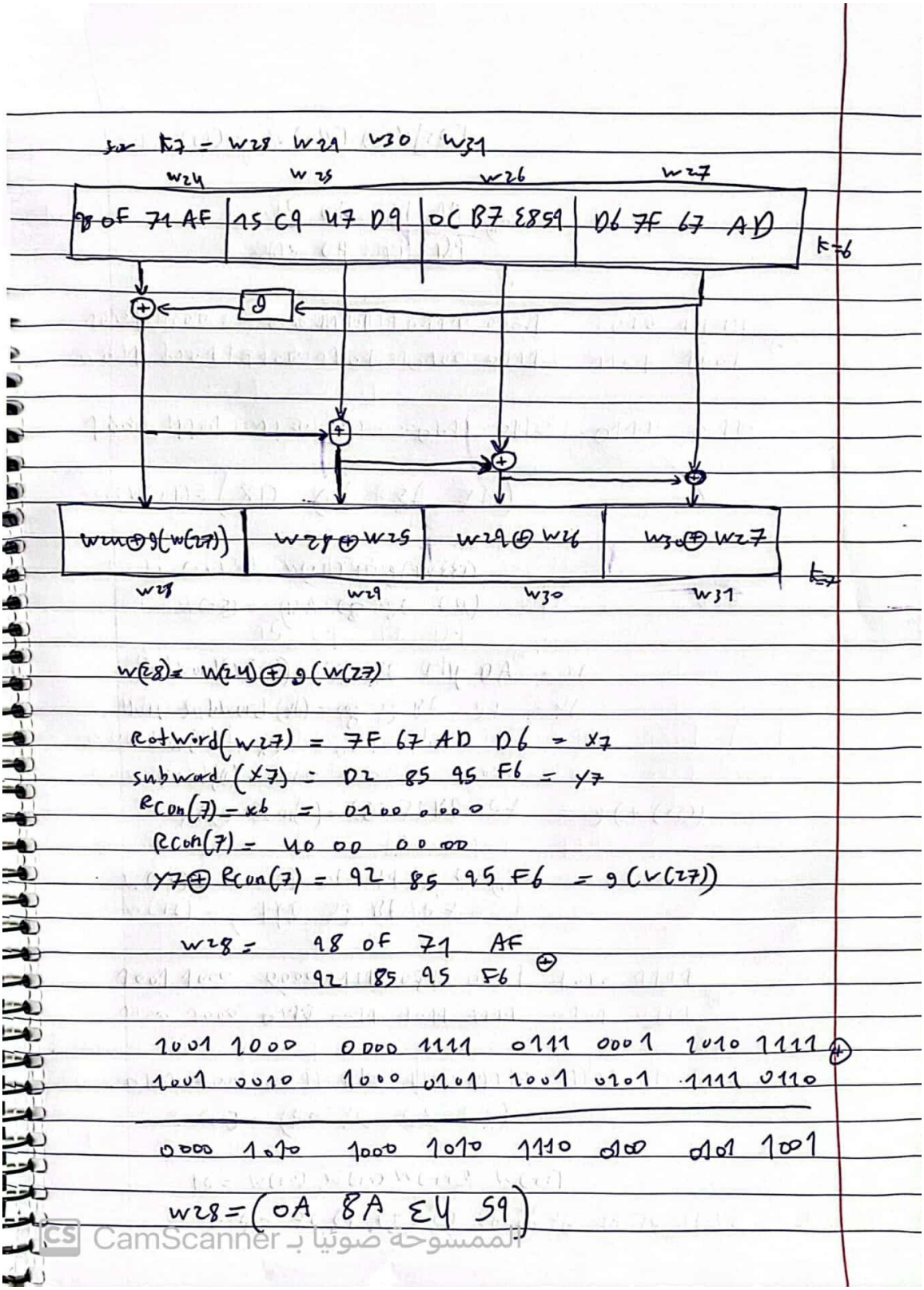
0101 0000 0111 1100 1100 1110 1111 1000

$w[20] = (50\ 7C\ CE\ F8)$

$ts = w[20]\ w[21]\ w[22]\ w[23]$

$= (50\ 7C\ CE\ F8\ 8D\ C6\ 36\ 76\ 19\ 7E\ AF\ 80\ DA\ C8\ 8F\ F4)$

For K7



$w(28) = w(24) \oplus g(w(27))$

RotWord$(w27)$ = 7F 67 AD D6 = x7

subword $(x7)$ = D2 85 95 F6 = y7

Rcon(7) = x6 = 0100 0000

Rcon(7) = 40 00 00 00

y7 $\oplus$ Rcon(7) = 92 85 95 F6 = g(w(27))

w28 = 98 0F 71 AF $\oplus$
92 85 95 F6

1001 1000 0000 1111 0111 0001 1010 1111 $\oplus$
1001 0010 1000 0101 1001 0101 1111 0110

0000 1010 1000 1010 1110 0100 0101 1001

w28 = (0A 8A E4 59)

$w[29] = w[28] \oplus w[25]$

0A 8A E4 59
15 C9 47 D9 ⊕

0000 1010 1000 1010 1110 0100 0101 1001
0001 0101 1100 1001 0100 0111 1101 1001 ⊕

0001 1111 0100 0011 1010 0011 1000 0000

$w[29] = (1F\ 43\ A3\ 80)$

$w[30] = w[29] \oplus w[26]$

1F 43 A3 80
0C B7 E8 59 ⊕

0001 1111 0100 0011 1010 0011 1000 0000
0000 1100 1011 0111 1110 1000 0101 1001 ⊕

0001 0011 1111 0100 0100 1011 1101 1001

$w[30] = (13\ F4\ 4B\ D9)$

$w[31] = w[30] \oplus w27$

13 F4 4B D9
D6 7F 67 AD ⊕

0001 0011 1111 0100 0100 1011 1101 1001
1101 0110 0111 1111 0110 0111 1010 1101 ⊕

1100 0101 1000 1011 0010 1100 0111 0100

$w[31] = (C5\ 8B\ 2C\ 74)$

So $b7 = (0A\ 8A\ E4\ 59\ 1F\ 43\ A3\ 80\ 13\ F4\ 4B\ D9\ C5\ 8B\ 2C\ 74)$

Question10

This question requires you to explore and evaluate the key cryptographic properties of substitution boxes (S-Boxes) used in symmetric-key cryptography. Focus on the following properties: (1) Bijection, (2) Nonlinearity, (3) Strict Avalanche Criterion (SAC), and (4) Output Bits Independence Criterion (BIC). Start by selecting a peer-reviewed research paper that examines these performance properties of cryptographic S-Boxes, and ensure you reference this paper in your submission. Clearly define each property in your own words, ensuring accuracy and clarity. Next, analyze the AES S-Box by measuring these properties, detailing your methodology step by step, and presenting your results using well-organized tables or graphs. Submit your work in both soft and hard copy formats.

1. Bijection

Explanation: A bijective S-Box mean every input byte goes to a unique output byte. This mean no two different inputs can have same output. This is very important for cryptography because it make sure encryption can reverse to get original message.

Mathematically: A function: $A \rightarrow B$ (f:A→B) is bijective if the size of A (|A|) is same as size of B (|B|). Also, every element in B (b∈B) comes from only one element in A (a∈A). This mean:

- If $(a) = (a')$, then $a = a'$. (injective)
- And, for every b in B, there is an a in A so f(a) =b . (subjective)

2. Nonlinearity

Explanation: Nonlinearity show how far the S-Box is from any simple linear function. If S-Box have high nonlinearity, it become hard to guess or attack using linear or differential methods. This is important for making S-Box strong in cryptography.

Mathematically: Nonlinearity is calculated by something called the Walsh-Hadamard transform. The nonlinearity (N f) is:

$Nf=2^{n-1} - 0.5 \max(|W f (\omega)|)$

Here, Wf (ω) measures how much the S-Box Boolean function is close to affine (linear-like) functions. High nonlinearity mean it is very far from linear.

3. Strict Avalanche Criterion (SAC)

Explanation: SAC mean when changing one bit in the input of S-Box, each output bit should change about 50% of the time. This is important for security because it make sure small change in input spread to many parts of the output. This help in making strong mixing (diffusion) in the cipher.

Mathematically: For input with n bits, SAC is checked by seeing if flipping one input bit changes each output bit with chance close to 0.5. Usually, we use a correlation matrix to check how input differences affect output bits.

4. Output Bits Independence Criterion (BIC)

Explanation: BIC is like SAC but go further. It mean when one input bit is changed, the changes in output bits should not depend on each other. This makes sure that knowing how one output bit change does not tell anything about other bits. It help to make cipher more secure.

Mathematically: To check BIC, we look at how different output bits change and see if they are independent. This needs more advance statistical methods to study how output changes happen together when input is changed.

Code Results:

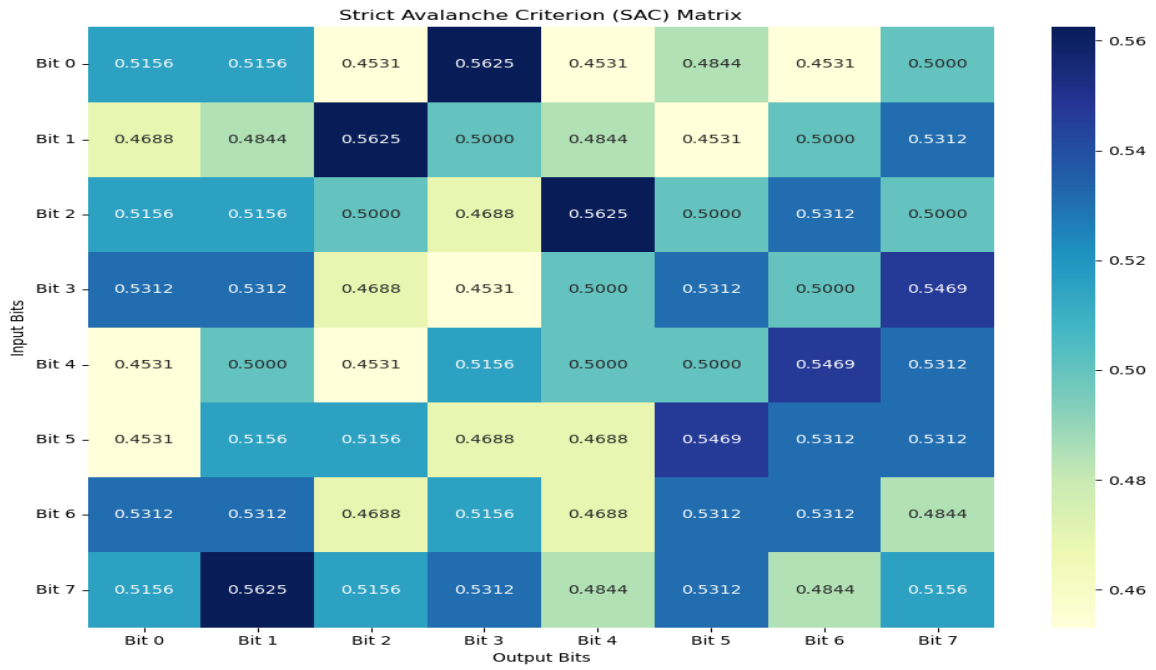
```
AES S-Box Cryptographic Properties Analysis

1. Bijection Check:
   - Result: True
   - Explanation: Every input goes to only one output. This show the S-Box is bijective.
```

****

```
2. Nonlinearity:
   - Minimum Nonlinearity: 112.0
   - Explanation: Big minimum nonlinearity make strong defense against linear and differential attacks.
```

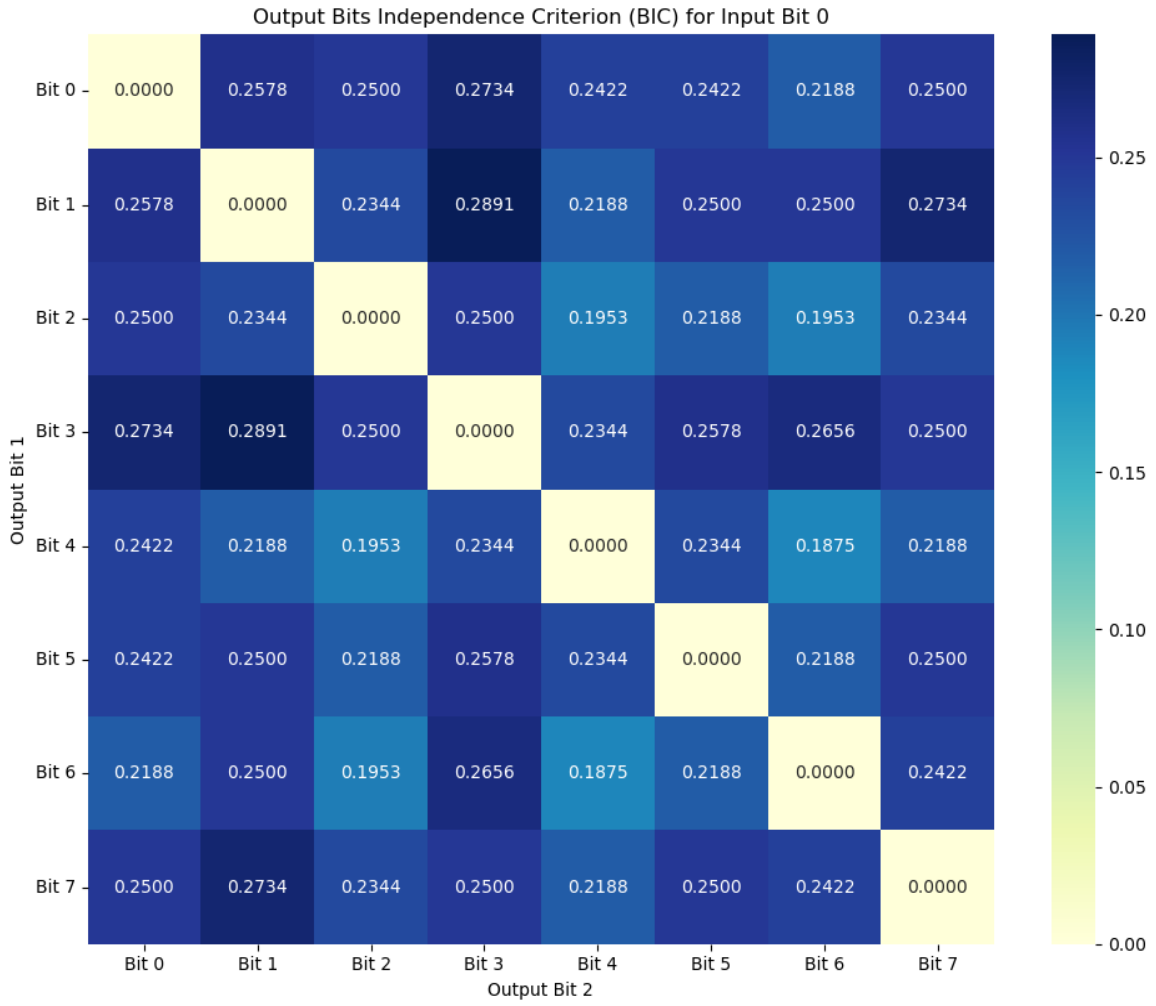
****

```
3. Strict Avalanche Criterion (SAC):
   - SAC Matrix:
          Bit 0   Bit 1   Bit 2   Bit 3   Bit 4   Bit 5   Bit 6   Bit 7
Bit 0 0.5156  0.5156  0.4531  0.5625  0.4531  0.4844  0.4531  0.5000
Bit 1 0.4688  0.4844  0.5625  0.5000  0.4844  0.4531  0.5000  0.5312
Bit 2 0.5156  0.5156  0.5000  0.4688  0.5625  0.5000  0.5312  0.5000
Bit 3 0.5312  0.5312  0.4688  0.4531  0.5000  0.5312  0.5000  0.5469
Bit 4 0.4531  0.5000  0.4531  0.5156  0.5000  0.5000  0.5469  0.5312
Bit 5 0.4531  0.5156  0.5156  0.4688  0.4688  0.5469  0.5312  0.5312
Bit 6 0.5312  0.5312  0.4688  0.5156  0.4688  0.5312  0.5312  0.4844
Bit 7 0.5156  0.5625  0.5156  0.5312  0.4844  0.5312  0.4844  0.5156

   - Explanation: Flipping one input bit change the output bits around 50% of times.
```



****

```
4. Output Bits Independence Criterion (BIC):
   - Sample BIC Results for Input Bit 0:
       Bit 0  Bit 1  Bit 2  Bit 3  Bit 4  Bit 5  Bit 6  Bit 7
Bit 0 0.0000  0.2578  0.2500  0.2734  0.2422  0.2422  0.2188  0.2500
Bit 1 0.2578  0.0000  0.2344  0.2891  0.2188  0.2500  0.2500  0.2734
Bit 2 0.2500  0.2344  0.0000  0.2500  0.1953  0.2188  0.1953  0.2344
Bit 3 0.2734  0.2891  0.2500  0.0000  0.2344  0.2578  0.2656  0.2500
Bit 4 0.2422  0.2188  0.1953  0.2344  0.0000  0.2344  0.1875  0.2188
Bit 5 0.2422  0.2500  0.2188  0.2578  0.2344  0.0000  0.2188  0.2500
Bit 6 0.2188  0.2500  0.1953  0.2656  0.1875  0.2188  0.0000  0.2422
Bit 7 0.2500  0.2734  0.2344  0.2500  0.2188  0.2500  0.2422  0.0000

   - Explanation: Output bits change separate when one input bit is flipped. The values need to be near 0.25.
```



****

References:

https://link.springer.com/chapter/10.1007/3-540-39799-X_41

https://iacr.org/cryptodb/data/paper.php?pubkey=1829

Code Screens:

Question10.py

```python
# Mohammad Abu Shams 1200549

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns


# Define the AES S-Box.
aes_sbox = np.array([
    [0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76],
    [0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0],
    [0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15],
    [0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75],
    [0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84],
    [0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf],
    [0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8],
    [0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2],
    [0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73],
    [0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb],
    [0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79],
    [0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08],
    [0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a],
    [0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e],
    [0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf],
    [0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16]
]).flatten()


# Function to check bijection.
def check_bijection(sbox):
    return len(set(sbox)) == 256


# Walsh-Hadamard transform.
def walsh_hadamard_transform(f, n):
    N = 2 ** n
    wht = f.copy()
    for i in range(n):
        step = 2 ** i
        for j in range(0, N, step * 2):
            for k in range(step):
                a = wht[j + k]
                b = wht[j + k + step]
                wht[j + k] = a + b
                wht[j + k + step] = a - b
    return wht


# Function to calculate nonlinearity using Walsh-Hadamard transform.
def calculate_nonlinearity(sbox):
    n = 8
    min_nonlinearity = float('inf')

    for output_bit in range(n):
        f = [(x >> output_bit) & 1 for x in sbox]
        f_mapped = [1 if bit == 0 else -1 for bit in f]
        wht = walsh_hadamard_transform(f_mapped, n)
        max_wht = max(np.abs(wht))
        nonlinearity = (2 ** (n - 1)) - (0.5 * max_wht)
        min_nonlinearity = min(min_nonlinearity, nonlinearity)

    return min_nonlinearity


# Function to calculate Strict Avalanche Criterion (SAC).
def calculate_sac(sbox):
    size = 256
    input_bits = 8
    output_bits = 8
    sac_matrix = np.zeros((input_bits, output_bits))

    for i in range(size):
        for bit in range(input_bits):
            flipped_input = i ^ (1 << bit)
            output_diff = sbox[i] ^ sbox[flipped_input]
            for output_bit in range(output_bits):
                sac_matrix[bit][output_bit] += (output_diff >> output_bit) & 1

    sac_matrix /= size
    return sac_matrix


```

```python
# Function to calculate Output Bits Independence Criterion (BIC).
def calculate_bic(sbox):
    size = 256
    input_bits = 8
    output_bits = 8
    bic_matrix = np.zeros((input_bits, output_bits, output_bits))

    for i in range(size):
        for bit in range(input_bits):
            flipped_input = i ^ (1 << bit)
            output_diff = sbox[i] ^ sbox[flipped_input]
            for output_bit1 in range(output_bits):
                for output_bit2 in range(output_bits):
                    if output_bit1 != output_bit2:
                        bic_matrix[bit][output_bit1][output_bit2] += (
                            ((output_diff >> output_bit1) & 1) *
                            ((output_diff >> output_bit2) & 1)
                        )

    bic_matrix /= size
    return bic_matrix


# Function to display SAC matrix as a heatmap.
def plot_sac_matrix(sac_matrix):
    plt.figure(figsize=(10, 8))
    sns.heatmap(sac_matrix, annot=True, fmt=".4f", cmap="YlGnBu")
    plt.title("Strict Avalanche Criterion (SAC) Matrix")
    plt.xlabel("Output Bits")
```

```python
    plt.ylabel("Input Bits")
    plt.xticks(np.arange(8)+0.5, [f"Bit {i}" for i in range(8)], rotation=0)
    plt.yticks(np.arange(8)+0.5, [f"Bit {i}" for i in range(8)], rotation=0)
    plt.tight_layout()
    plt.show()


# Function to display BIC matrix for a specific input bit as a heatmap.
def plot_bic_matrix(bic_matrix, input_bit):
    plt.figure(figsize=(10, 8))
    sns.heatmap(bic_matrix[input_bit], annot=True, fmt=".4f", cmap="YlGnBu")
    plt.title(f"Output Bits Independence Criterion (BIC) for Input Bit {input_bit}")
    plt.xlabel("Output Bit 2")
    plt.ylabel("Output Bit 1")
    plt.xticks(np.arange(8)+0.5, [f"Bit {i}" for i in range(8)], rotation=0)
    plt.yticks(np.arange(8)+0.5, [f"Bit {i}" for i in range(8)], rotation=0)
    plt.tight_layout()
    plt.show()


# MAIN.
def main():
    print("AES S-Box Cryptographic Properties Analysis\n")

    # 1. Bijection Check.
    bijection = check_bijection(aes_sbox)
    print("1. Bijection Check:")
    print(f"   - Result: {bijection}")
    print("   - Explanation: Every input goes to only one output. This show the S-Box is bijective.\n")
```

```python


    # 2. Nonlinearity Calculation.
    nonlinearity = calculate_nonlinearity(aes_sbox)
    print("2. Nonlinearity:")
    print(f"   - Minimum Nonlinearity: {nonlinearity}")
    print("   - Explanation: Big minimum nonlinearity make strong defense against linear and differential attacks.\n")


    # 3. Strict Avalanche Criterion (SAC) Calculation.
    sac = calculate_sac(aes_sbox)
    print("3. Strict Avalanche Criterion (SAC):")
    print("   - SAC Matrix:")
    # Formatting the SAC matrix for display.
    sac_table = ""
    header = "      " + "  ".join([f"Bit {i}" for i in range(8)])
    sac_table += header + "\n"
    for i, row in enumerate(sac):
        sac_table += f"Bit {i} " + "  ".join([f"{val:.4f}" for val in row]) + "\n"
    print(sac_table)
    # Plotting the SAC matrix heatmap.
    plot_sac_matrix(sac)
    print("   - Explanation: Flipping one input bit change the output bits around 50% of times.\n")


```

```
    # 4. Output Bits Independence Criterion (BIC) Calculation.
    bic = calculate_bic(aes_sbox)
    print("4. Output Bits Independence Criterion (BIC):")
    print("   - Sample BIC Results for Input Bit 0:")
    # Formatting the BIC matrix for input bit 0.
    bic_table = ""
    header = "       " + "   ".join([f"Bit {i}" for i in range(8)])
    bic_table += header + "\n"
    for i, row in enumerate(bic[0]):
        bic_table += f"Bit {i} " + "   ".join([f"{val:.4f}" for val in row]) + "\n"
    print(bic_table)
    # Plotting the BIC matrix heatmap for input bit 0
    plot_bic_matrix(bic, 0)
    print("   - Explanation: Output bits change separate when one input bit is flipped. The values need to be near 0.25.\n")


if __name__ == "__main__":
    main()

```

Appendix:

```
# Mohammad Abu Shams 1200549

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns


# Define the AES S-Box.
aes_sbox = np.array([
    [0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b,
0xfe, 0xd7, 0xab, 0x76],
    [0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf,
0x9c, 0xa4, 0x72, 0xc0],
    [0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1,
0x71, 0xd8, 0x31, 0x15],
    [0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2,
0xeb, 0x27, 0xb2, 0x75],
    [0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3,
0x29, 0xe3, 0x2f, 0x84],
    [0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39,
0x4a, 0x4c, 0x58, 0xcf],
    [0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f,
0x50, 0x3c, 0x9f, 0xa8],
    [0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21,
0x10, 0xff, 0xf3, 0xd2],
    [0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d,
0x64, 0x5d, 0x19, 0x73],
    [0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14,
0xde, 0x5e, 0x0b, 0xdb],
    [0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62,
0x91, 0x95, 0xe4, 0x79],
    [0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea,
0x65, 0x7a, 0xae, 0x08],
    [0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f,
0x4b, 0xbd, 0x8b, 0x8a],
    [0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9,
0x86, 0xc1, 0x1d, 0x9e],
    [0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9,
```

```
0xce, 0x55, 0x28, 0xdf],
    [0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f,
0xb0, 0x54, 0xbb, 0x16]
]).flatten()


# Function to check bijection.
def check_bijection(sbox):
    return len(set(sbox)) == 256


# Walsh-Hadamard transform.
def walsh_hadamard_transform(f, n):
    N = 2 ** n
    wht = f.copy()
    for i in range(n):
        step = 2 ** i
        for j in range(0, N, step * 2):
            for k in range(step):
                a = wht[j + k]
                b = wht[j + k + step]
                wht[j + k] = a + b
                wht[j + k + step] = a - b
    return wht


# Function to calculate nonlinearity using Walsh-Hadamard transform.
def calculate_nonlinearity(sbox):
    n = 8
    min_nonlinearity = float('inf')

    for output_bit in range(n):
        f = [(x >> output_bit) & 1 for x in sbox]
        f_mapped = [1 if bit == 0 else -1 for bit in f]
        wht = walsh_hadamard_transform(f_mapped, n)
        max_wht = max(np.abs(wht))
        nonlinearity = (2 ** (n - 1)) - (0.5 * max_wht)
        min_nonlinearity = min(min_nonlinearity, nonlinearity)

    return min_nonlinearity


# Function to calculate Strict Avalanche Criterion (SAC).
def calculate_sac(sbox):
    size = 256
    input_bits = 8
    output_bits = 8
    sac_matrix = np.zeros((input_bits, output_bits))

    for i in range(size):
        for bit in range(input_bits):
            flipped_input = i ^ (1 << bit)
            output_diff = sbox[i] ^ sbox[flipped_input]
            for output_bit in range(output_bits):
                sac_matrix[bit][output_bit] += (output_diff >> output_bit) &
1
```

```
    sac_matrix /= size
    return sac_matrix


# Function to calculate Output Bits Independence Criterion (BIC).
def calculate_bic(sbox):
    size = 256
    input_bits = 8
    output_bits = 8
    bic_matrix = np.zeros((input_bits, output_bits, output_bits))

    for i in range(size):
        for bit in range(input_bits):
            flipped_input = i ^ (1 << bit)
            output_diff = sbox[i] ^ sbox[flipped_input]
            for output_bit1 in range(output_bits):
                for output_bit2 in range(output_bits):
                    if output_bit1 != output_bit2:
                        bic_matrix[bit][output_bit1][output_bit2] += (
                            ((output_diff >> output_bit1) & 1) *
                            ((output_diff >> output_bit2) & 1)
                        )

    bic_matrix /= size
    return bic_matrix


# Function to display SAC matrix as a heatmap.
def plot_sac_matrix(sac_matrix):
    plt.figure(figsize=(10, 8))
    sns.heatmap(sac_matrix, annot=True, fmt=".4f", cmap="YlGnBu")
    plt.title("Strict Avalanche Criterion (SAC) Matrix")
    plt.xlabel("Output Bits")
    plt.ylabel("Input Bits")
    plt.xticks(np.arange(8)+0.5, [f"Bit {i}" for i in range(8)], rotation=0)
    plt.yticks(np.arange(8)+0.5, [f"Bit {i}" for i in range(8)], rotation=0)
    plt.tight_layout()
    plt.show()


# Function to display BIC matrix for a specific input bit as a heatmap.
def plot_bic_matrix(bic_matrix, input_bit):
    plt.figure(figsize=(10, 8))
    sns.heatmap(bic_matrix[input_bit], annot=True, fmt=".4f", cmap="YlGnBu")
    plt.title(f"Output Bits Independence Criterion (BIC) for Input Bit
{input_bit}")
    plt.xlabel("Output Bit 2")
    plt.ylabel("Output Bit 1")
    plt.xticks(np.arange(8)+0.5, [f"Bit {i}" for i in range(8)], rotation=0)
    plt.yticks(np.arange(8)+0.5, [f"Bit {i}" for i in range(8)], rotation=0)
    plt.tight_layout()
    plt.show()


# MAIN.
def main():
    print("AES S-Box Cryptographic Properties Analysis\n")
```

```
    # 1. Bijection Check.
    bijection = check_bijection(aes_sbox)
    print("1. Bijection Check:")
    print(f"   - Result: {bijection}")
    print("   - Explanation: Every input goes to only one output. This show
the S-Box is bijective.\n")


    # 2. Nonlinearity Calculation.
    nonlinearity = calculate_nonlinearity(aes_sbox)
    print("2. Nonlinearity:")
    print(f"   - Minimum Nonlinearity: {nonlinearity}")
    print("   - Explanation: Big minimum nonlinearity make strong defense
against linear and differential attacks.\n")


    # 3. Strict Avalanche Criterion (SAC) Calculation.
    sac = calculate_sac(aes_sbox)
    print("3. Strict Avalanche Criterion (SAC):")
    print("   - SAC Matrix:")
    # Formatting the SAC matrix for display.
    sac_table = ""
    header = "      " + "  ".join([f"Bit {i}" for i in range(8)])
    sac_table += header + "\n"
    for i, row in enumerate(sac):
        sac_table += f"Bit {i} " + "  ".join([f"{val:.4f}" for val in row]) +
"\n"
    print(sac_table)
    # Plotting the SAC matrix heatmap.
    plot_sac_matrix(sac)
    print("   - Explanation: Flipping one input bit change the output bits
around 50% of times.\n")


    # 4. Output Bits Independence Criterion (BIC) Calculation.
    bic = calculate_bic(aes_sbox)
    print("4. Output Bits Independence Criterion (BIC):")
    print("   - Sample BIC Results for Input Bit 0:")
    # Formatting the BIC matrix for input bit 0.
    bic_table = ""
    header = "      " + "  ".join([f"Bit {i}" for i in range(8)])
    bic_table += header + "\n"
    for i, row in enumerate(bic[0]):
        bic_table += f"Bit {i} " + "  ".join([f"{val:.4f}" for val in row]) +
"\n"
    print(bic_table)
    # Plotting the BIC matrix heatmap for input bit 0
    plot_bic_matrix(bic, 0)
    print("   - Explanation: Output bits change separate when one input bit
is flipped. The values need to be near 0.25.\n")


if __name__ == "__main__":
    main()
```