Q1 Team Name
0 Points

Group Name

d2ce09fd5842b342d5b3b66d10b6daef

Q2 Commands
5 Points

List all the commands in sequence used from the start screen of this level to the end of the level

5 -> go -> wave -> dive -> go -> read -> utomcrlsfj

Q3 Cryptosystem
5 Points

What cryptosystem was used at this level?

EAEAE cipher (a variant of AES cipher) - a weak form of SASAS attack. Where "EAEAE" cipher stands for "Extended Hexadecimal Ascii Encryption" cipher.

Q4 Analysis
80 Points

Knowing which cryptosystem has been used at this level, give a detailed description of the cryptanalysis used to figure out the password.

The Steps We used to do cryptanalysis are as follows :
part A: Explanation :
(i) We gathered ourselves in front of a passage which lead to a deep underground well, so
our resolution was to go ahead by putting the

command **"go."**

(ii) However, on the subsequent screen, we had a free fall without anything to hold. We wrote the **"go"** command again, it failed.

(iii) We tried again and used the command **"wave"** in place of "go" on the second screen, and this time we were victorious in reaching the third(next) screen.

(iv) On the third screen, we were swimming in both directions but could not find something of curiosity. In an endeavour to go ahead in positive direction , we utilized the **"dive"** command.

(v) This was the reason for us to be in the next screen, where we came across a well-lit passage in the wall. We went ahead by utilizing the **"go"** command.

(vi) The next screen had a glass panel, on which we utilized the **"read"** command. Not having a good luck, we came across a problem referred to as the **"EAEAE problem."**

(vii). Upon deeper examination, we found out that the problem had input with a block size of 8 bytes, represented as an 8 x 1 vector over the finite field F_{128}.

(viii) The EAEAE is a minuscule form of SASAS attack. Differential cryptanalysis attacks that take advantage of block cypher flaws include SASAS and EAEAE. These attacks include comparing the differences between plaintexts and the ciphertexts produced by the encryption procedure in an effort to discover the encryption key or to decrypt the data. The block cypher used by the EAEAE encryption algorithm runs on a fixed-size block of 8 bytes. Each byte is an element of the finite field F_{128}, which consists of 128 elements represented as 16-byte vectors over the binary field GF(2). The field F_{128} is defined by the irreducible polynomial $x^{128} + x^7 + x^2 + x + 1.$

(ix) When plaintext is has encryption by virtue of EAEAE, the corresponding ciphertext comprise of 8 bytes, each of which is an element of F_{128}. However, our analysis of ciphertexts generated by EAEAE, we came under the impression that the output contained only 16 different letters ranging from **'f' to 'u'**.

part A : Initial Analysis

1. Our first task was to get hold of the encoding scheme employed for converting the string (That follows will be utilized as an input sequence) into the blocks of 0's and 1' s Encoding.
The first thought which hit our mind was that first thing we should check could obviously be the ASCII values. So, we begin our mission of analyzing the output. The thing which we encountered while judging the output was that it belonged to a range of alphabetical literals from letter 'f' to 'u'. Since the count from 'f' to 'u' is 16 we got an anticipation that it has some connection with the hexadecimal system which has a base of 16. Where the odd positions contain the characters from 'f' to 'm' whereas the even positions contain the characters from 'f' to 'u'.

2. After giving numerous plaintext inputs to the encryption we observed a surprising thing in
the ciphertext that it accommodated only 16 letters from 'f' to 'u' . Our next decision was to
represent letter 'f' with 4 bits 0000, 'g' with 0001 , h with 0002 and so on going up to letter 'u'
with 1111 , since 4 bits can have 2^4 =16 combinations so we decided to map our cipher text
letters with 4 bit binary. This leads to a further deduction that a byte is comprised of 2 letters .

3. Also We have a prior knowledge that each byte is an element of F-128 in range of 0 to 127, So
the MSB of each byte should be 0. So, the possible byte value will range from 00000000 till

01111111 which has a representation in letters as from 'ff' to 'mu' . So the possible letter
pairs which are in our use are from 'ff' to 'mu' . This research reveals a flaw in the EAEAE
encryption algorithm that enables the ciphertext to be expressed using a smaller number of
symbols than anticipated, making it susceptible to attacks.

part B : Cryptanalysis

4. Some Study/Analysis Done :
(i) If all the bits of input plaintext are zeros i.e when we input ffffffffffffffff as plaintext the to
our surprise we get again ffffffffffffffff as output .

(ii) if first 'x' bytes of plaintext is comprised only of f's then the corresponding ciphertext will
also have first 'x' bytes as all f's .

(iii) Our next move was to change the i th byte of plaintext and we got the result that uptill
i th byte the ciphertext was unchanged. Let the input plaintext be $P_0, P_1, P_2, ..., P_7$, where
Pi is comprised of 1 byte. If we now change the input plaintext from $P_0, P_1, .., P_k, P_{k+1}, ..., P_7$.
then the result we get will be changed from kth byte onwards, then resulting ciphertext will
differ from kth byte onwards.

(iv) This Observation gave us the intuition that the matrix A is a lower triangular matrix.

5. Strategies employed to calculate the transformation matrices A and E .

(i) With the above observation in hand, it becomes fairly simple to crack the Exponential
transformation box E and diagonal elements of the Linear Transform A. The matrix A is of
dimensions 8*8 and E is of dimensions 8*1. where a ij belongs to A, where i is the row
index and j is the column index and e i is the ith

element of to E .

(ii) To generate the the plaintext employed in the attack we used the file plaintext.py.

To generate different plaintext we used the formula $C^i P C^{8-i}$. Where C='ff', and P belongs to [ff, mu]. i belongs to [1,8] . Utilizing 8 Sets of plaintext containing 128 plaintext each where in each set i ,the i^{th} byte's are differed These plaintext are stored in the plaintext.txt .

(iii) The cipher-texts for the plain-texts of the type where just one block is non-zero are first fetched. 127 different plain-texts, numbered from 1 to 127, are possible for each block option. One can iterate over the selection of diagonal elements and the exponentiation corresponding to the non-zero block using these 127 plain-text ciphertext pairs.

(iv) One basic observation is, given the linear transform is lower triangular matrix, any block of output, let's say i_{th} block, is dependent on j_{th} block of input iff i ≥ j.

In our chosen pattern for input, if x is the value of non-zero input block(say i), then the corresponding block of output has the value $(a_{i,i}(a_{i,i}*x^{ei})^{ei})^{ei}$.

Iterating over values of ai,i (from 0 to 127) and ei (from 1 to 126), one gets 3 tuples of values for each block.

$$E^{-1}(A^{-1}(E^{-1}(A^{-1}(E^{-1}(p)))))$$

(iii) Each plaintext from the file we generated previously was given as input to file robot.py to generate corresponding Ciphertext . robot.py is python script using python library 'pexpect' to establish a connection to the game server , input commands order by order and pass plaintext to generate the corresponding ciphertext. These obtained ciphertext are stored in file ciphertext.txt.

(iv) We have a prior knowledge that A is a lower triangular matrix and C is

$$E^{-1}(A^{-1}(E^{-1}(A^{-1}(E^{-1}(p)))))$$

Where C is ciphertext and P is plaintext. Our next step in this investigation is that to get hold of the possible diagonal elements of matrix A and find the elements of E with the help of brute force strategies .

(v) The encryption process is performing the following operations in order :
Linear Transformation , Exponentiation operation , Linear Transformation ,
Exponentiation over the Finite Field F 128 over the irreducible polynomial x^7 + x + 1,
which is irreducible polynomial over F 2 is utilized to perform the operations . The addition operation is implemented as Exclusive -OR(XOR) of integers in the F128 finite field.

(vi) To get hold of the diagonal values of matrix A and E, for each plaintext - ciphertext pair We iterate over the values [0,127] and [1,126] for E to check whether the plaintext on encryption map to the corresponding ciphertext or not . We now store the values where ciphertext map to plaintext in the table given below:

(i^{th}) Byte	($a_{i,i}$) Possible Values	(e_i) Possible Values
0	[73, 84, 20]	[18, 21, 88]
1	[122, 62, 70]	[26, 113, 115]
2	[119, 43, 5]	[2, 38, 87]
3	[12, 52, 100]	[71, 79, 104]
4	[5, 31, 112]	[78, 85, 91]
5	[11, 122, 100]	[45, 93, 116]
6	[52, 26, 27]	[1, 19, 107]
7	[38, 65, 52]	[22, 37, 68]

(vii) Next we needed to know the values of non-diagonal

elements of matrix A and get rid of
some pairs of (a ii,ei) . We perform a iteration over the plaintext - ciphertext
pairs and try to find values from which equation 1 which we mention in point 5(iv) can
get satisfied. When the jth block of input is non-zero, any element ai,j can be found by
examining the ith block of output. In addition to the diagonal elements, several more
elements of A are required to calculate this.

(viii) To find aij we have to know all values of the set
Si,j = {an,m|n > m, j ≤ n, m ≤ i} ∩ {an,n|j ≤ n ≤ i}.

(ix) These components can be seen as forming a right-angled triangle with the corners aj,j,
ai,j, and ai,i. Iterative search for non-diagonal is conducted. ai+1,i elements are discussed
first, then ai+2,i, and so forth. We could remove the superfluous pairs from the
aforementioned matrix while looking for the value of ai+1, i.

i^{th} Byte	values of $a_{i,i}$	values of e_i
0	84	113
1	70	30
2	43	1
3	12	109
4	112	98
5	11	95
6	27	0
7	38	38

(x) We needed 127*8 selected plain-texts in total to complete the assault. Although it appears
possible, we haven't looked into the potential of achieving this with less input to the
encryption engine. The necessary operations are 128*128*36 (2^19). (Matrix has 36 non-
zero elements, and the most operations needed to find any element are 128*128).

(xi) Alternative Multiply and Exponentiate functions have been used for implementation. The
XOR operation in integers is analogous to the addition in F128. The two examples above
were adjusted appropriately, and the values were put into 128*128 matrices. We then
utilised these to build our Encrypt function, which was used to compare our encrypted
output to the real encrypted output during the brute force attack on each of the ai, j, and
ek.

Final Linear transformation matrix A is

$$\begin{bmatrix} 84 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 113 & 70 & 0 & 0 & 0 & 0 & 0 & 0 \\ 13 & 30 & 43 & 0 & 0 & 0 & 0 & 0 \\ 100 & 23 & 11 & 12 & 0 & 0 & 0 & 0 \\ 110 & 34 & 1 & 109 & 112 & 0 & 0 & 0 \\ 25 & 44 & 28 & 53 & 98 & 11 & 0 & 0 \\ 2 & 122 & 23 & 103 & 27 & 95 & 27 & 0 \\ 65 & 2 & 86 & 27 & 15 & 66 & 0 & 38 \end{bmatrix}$$

Final Exponent vector E is below:

$$\mathbf{E} = [\mathbf{21, 115, 38, 71, 91, 45, 19, 22}]$$

6. Password Decryption Process :

(i) Now since we have successfully found the matrix A and matrix E we can find the password
by applying reverse transformations for each 8 byte block of encrypted password p ,We
carry out following operations to obtain the 8 byte decrypted password :

(ii) Our encrypted password which we got is 'lhgomlmjmohhmkmpjqkmksmngnglflif'.

Encrypted block 1 = 'lhgomlmjmohhmkmp'
Encrypted block 2 = 'jqkmksmngnglflif'

Decrypted Block 1 ASCII : [117, 116, 111, 109, 99, 114, 108, 115]
Decryted Password 1 : 'utomcrls'

Decrypted Block 2 ASCII: [102, 106, 48, 48, 48, 48, 48, 48]
Decryted Password 2 : 'fj000000'

Decrypted Password: 'utomcrlsfj000000'

We assumed 000000 as padding at the end and we tried this password for the level and successfully cleared it.

7. Elaborate reason for our choice of Lower Triangular Matrix:

(i) The terms we utilized in our analysis are as follows :
• $[p0, p1, p2, p3, p4, p5, p6, p7]$ is the input 8 byte plaintext.
• $[c0, c1, c2, c3, c4, c5, c6, c7]$ is the output 8 byte ciphertext.
• 0 represents 'ff'
• The linear transform matrix is represented as

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & \cdots & a_{0,6} & a_{0,7} \\ a_{1,0} & a_{1,1} & a_{1,2} & \cdots & a_{1,6} & a_{1,7} \\ a_{2,0} & a_{2,1} & a_{2,2} & \cdots & a_{2,6} & a_{2,7} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{6,0} & a_{6,1} & a_{6,2} & \cdots & a_{6,6} & a_{6,7} \\ a_{7,0} & a_{7,1} & a_{7,2} & \cdots & a_{7,6} & a_{7,7} \end{bmatrix}$$

(ii) We discovered that the ciphertexts had the pattern ci = 0 for i < k by giving input where p_k is altered while keeping other p_i = 0. The output will be a vector if the vector [x0 x1 x2 ... x7] is the input to the final A layer.

$$\begin{bmatrix} y_0 = a_{0,0} \cdot x_0 + a_{0,1} \cdot x_1 + a_{0,2} \cdot x_2 + \cdots + a_{0,7} \cdot x_7 \\ y_1 = a_{1,0} \cdot x_0 + a_{1,1} \cdot x_1 + a_{1,2} \cdot x_2 + \cdots + a_{1,7} \cdot x_7 \\ \vdots \\ y_7 = a_{7,0} \cdot x_0 + a_{7,1} \cdot x_1 + a_{7,2} \cdot x_2 + \cdots + a_{7,7} \cdot x_7 \end{bmatrix}$$

(iii) Any unique output matches a unique input since the final E layer is essentially a bijective mapping. A further advantage of the exponentiation process is that 0 at the output will correspond to 0 at the input. On passing input as 00 . . . pkpk+1 . . . p7 we obtained output as 00 . . . ckck+1 . . . c7. In other words, whenever the input plaintext has p0, .., pi = 0 the output will have c0, .., ci = 0.
This implies that input to the last E layer will also have been of the same format
00 . . . c'kc'k+1 . . . c'7. Thus the equation for yi = 0 had 249−7∗i solutions. This is only possible if all but i of ai,j is nonzero and the rest are 0.
(iv) Thus the first row will contain 7 0 elements, 2nd row 6 0 elements, 3rd 5 o elements and so on. We also observed that passing p0p1 . . . pkpk+1 . . . p7 and p0p1 . . . pkp'k+1 . . . p7, the output only changed after k, this implies that all the 0 present in each row have to be at the end of the row. Thus we get a lower triangular matrix.

Q5 Password
10 Points

What was the password used to clear this level?

utomcrlsfj

Q6 Code
0 Points

Please add your code here. It is MANDATORY.

AES_assignment5.ipynb Download

In [1]:

```
import numpy as np
from pyfinite import ffield
import galois

F = ffield.FField(7, gen=0x83, useLUT=-

def Exponentiate(base,power):
    ans = base
    for i in range(1,power):
        ans = F.Multiply(ans,base)
    return ans


def LinearTransform(linmat,msg):
    ans = [0]*8
    for i in range(8):
        temp  = []
        mul=[]

        for j in range(8):

mul.append(F.Multiply(linmat[i]
[j],msg[i]))

        for k in range(8):

temp.append(np.bitwise_xor(ans[k],mul[k
        ans = temp
    return ans
```

In [2]:

```
def dec_blk(cipher):
  plain= ""
  for i in range(0,len(cipher),2):
      plain+=chr(16*
(ord(cipher[i:i+2][0]) - ord('f'))
+ ord(cipher[i:i+2][1]) -
ord('f'))
  return plain
```

In [4]:

```
#for diagonal elements
PossibleExponents = [[] for i in range(
possibleDiagonalVals=[[[] for i in rang
for j in range(8)]
input_file = open('plaintexts.txt','r')
output_file = open('ciphertexts.txt','r
input = (input_file.readlines()
[0]).strip().split(' ')
```

```
output = output_file.readlines()

input_string = []
for msg in input:
    input_string.append(dec_blk(msg)[0]

output_string = []
for i in range(len(output)):
    x = []
    for msg in output[i].strip().split(
        x.append(dec_blk(msg)[i])
    output_string.append(x)
#print(output_string)
for k in range(8):
    for i in range(1, 127):
        for j in range(1, 128):
          flag = True
          for m in range(128):
            if(ord(output_string[k][m])
Exponentiate(F.Multiply(Exponentiate(F.
i), j), i), j), i)):
              flag = False
              break
          if(flag):
            PossibleExponents[k].append
            possibleDiagonalVals[k]
[k].append(j)
print("Possible diagonal values: \n")
print(possibleDiagonalVals)
print("\n\nPossible exponents: \n")
print(PossibleExponents)


output_string = []
for i in range(len(output)-1):
    x = []
    for msg in output[i].strip().split(
        x.append(dec_blk(msg)[i+1])
    output_string.append(x)

for ind in range(7):
  for i in range(1, 128):
      for p1, e1 in
zip(PossibleExponents[ind+1],
possibleDiagonalVals[ind+1][ind+1]):
          for p2, e2 in
zip(PossibleExponents[ind],
possibleDiagonalVals[ind][ind]):
              for k in range(128):
                  flag = True
                  x1 =
F.Multiply(Exponentiate(F.Multiply(Expo
```

```
p2), e2), p2), i)
                        x2 =
F.Multiply(Exponentiate(F.Multiply(Expo
p2), i), p1), e1)
                        c1 = np.bitwise_xor(x
                        if(ord(output_string[
!= Exponentiate(c1,p1)):
                             flag = False
                             break
                   if flag:
                        PossibleExponents[ind
[p1]
                        possibleDiagonalVals[
[ind+1] = [e1]
                        PossibleExponents[ind
                        possibleDiagonalVals[
[ind] = [e2]
                        possibleDiagonalVals[
[ind+1] = [i]
print('\n\n===========================\
print("Diagonal values: \n")
print(possibleDiagonalVals)
print("\n\nExponents: \n")
print(PossibleExponents)
```

```
Possible diagonal values:

[[[73, 84, 20], [], [], [], [], [], [],


Possible exponents:

[[18, 21, 88], [26, 113, 115], [2, 38,



===========================



Diagonal values:

[[[84], [113], [], [], [], [], [], []],


Exponents:

[[21], [115], [38], [71], [91], [45], [
```

In [5]:

```
def EAEAE (msg, lin_mat, exp_mat):
  msg = [ord(m) for m in msg]
  exponents = [Exponentiate(msg[i],
```

```
exp_mat[i]) for i in range(len(msg))]
  linear_transformed =
LinearTransform(lin_mat, exponents)
  exponents_2 =
[Exponentiate(linear_transformed[i],
exp_mat[i]) for i in
range(len(linear_transformed))]
  linear_transformed_2 =
LinearTransform(lin_mat, exponents_2)
  final_output =
[Exponentiate(linear_transformed_2[i],
exp_mat[i]) for i in
range(len(linear_transformed_2))]
  return final_output



input_file =
open('plaintexts.txt','r')
output_file =
open('ciphertexts.txt','r')
input = input_file.readlines()
output = output_file.readlines()



input_string = [[dec_blk(msg) for msg
in input[i].strip().split(' ')] for i
in range(len(input))]
output_string = [[dec_blk(msg) for msg
in output[i].strip().split(' ')] for
i in range(len(output))]

for indexex in range(0,6):
    offset = indexex + 2

    exp_list = [e[0] for e in
PossibleExponents]
    lin_trans_list =
np.zeros((8,8),int)

    for i in range(8):
      for j in range(8):
        if(len(possibleDiagonalVals[i]
[j]) != 0):
            lin_trans_list[i][j] =
possibleDiagonalVals[i][j][0]
        else:
            lin_trans_list[i][j] = 0

    for index in range(8):
        if(index > (7-offset)):
```

```
            continue

        for i in range(127):
            lin_trans_list[index]
[index+offset] = i+1
            flag = True
            for inps, outs in
zip(input_string[index],
output_string[index]):
                x1 = EAEAE(inps,
lin_trans_list, exp_list)
[index+offset]
                x2 =
outs[index+offset]
                if x1 != ord(x2):
                    flag = False
                    break
            if flag==True:

possibleDiagonalVals[index]
[index+offset] = [i+1]

A = np.zeros((8,8),dtype='int')

for i in range(0,8):
    for j in range(0,8):
      if len(possibleDiagonalVals[j]
[i]) != 0:
       A[i][j] =
possibleDiagonalVals[j][i][0]



E = exp_list

print('Linear Transformation Matrix
:\n',A)
print('\n\n')
print('Exponent Vector : \n',E)
```

```
Linear Transformation Matrix :
 [[ 84   0   0   0   0   0   0   0]
 [113  70   0   0   0   0   0   0]
 [ 13  30  43   0   0   0   0   0]
 [100  23   1  12   0   0   0   0]
 [110  34   1 109 112   0   0   0]
 [ 25  44  28  53  98  11   0   0]
 [  2 122  23 103  27  95  27   0]
 [ 65   2  86  27  15  66   0  38]]
```

```
Exponent Vector :
 [21, 115, 38, 71, 91, 45, 19, 22]
```

In [6]:

```
E_inverse = np.zeros((128, 128),
dtype = int)

for base in range(0,128):
    temp = 1
    for power in range(1,127):
        result = F.Multiply(temp,
base)
        E_inverse[power][result] =
base
        temp = result

GF = galois.GF(2**7)
A = GF(A)
invA = np.linalg.inv(A)
```

In [7]:

```
password =
"lhgomlmjmohhmkmpjqkmksmngnglflif"
#Encrypted password
GF = galois.GF(2**7)

def E_inv(block, E):
    new_list = []
    for i in range(8):
        new_list.append(E_inverse[E[i]]
[block[i]])
    return new_list

def A_inv(block, A):
    return np.matmul(GF(A),GF(block))

dec_pass = ""
for i in range(0,2):
    elements = password[16*i:16*(i+1)]
    currentBlock = []
    currentBlock += [(ord(elements[j])
ord('f'))*16 + (ord(elements[j+1]) -
ord('f')) for j in range(0, 15, 2)]

    EAEAE =
E_inv(A_inv(E_inv(A_inv(E_inv(currentBl
E), invA),E), invA),E)
    for ch in EAEAE:
        dec_pass += chr(ch)
```

```
print("\n\nPassword is",dec_pass)
```

```
Password is utomcrlsfj000000
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

aes_assignment5.zip Download

```
1  Binary file hidden. You can download it using the button above.
```

Assignment 5

Graded

Group

GUNJ MEHUL HUNDIWALA
RAJ KUMAR
MADHAV MAHESHWARI
View or edit group

Total Points

100 / 100 pts

Question 1

Team Name 0 / 0 pts

Question 2

Commands **5** / 5 pts

Question 3

Cryptosystem **5** / 5 pts

Question 4

Analysis **80** / 80 pts

Question 5

Password **10** / 10 pts

Question 6

Code **0** / 0 pts