Lab 04
Block Ciphers

Objective:

Stream ciphers work by generating pseudorandom bits and XORing them with your message. Block ciphers take in a fixed-length message, a private key, and they produce a ciphertext that is the same length as the fixed-length plaintext message. In this Lab students will understand working of DES, Triple DES AES.

Activity Outcomes:

- How to learn coding of cryptographic block ciphers like DES, and AES.
- How to code and analyze attack scenarios for complex block ciphers.

Sr.No	***Allocated Time***	***Level of Complexity***	***CLO Mapping***
1	***80***	***High***	***CLO-6***
2	***40***	***Medium***	
3	***50***	***High***	

Practice -Lab Activity 1: DES

AES and Triple DES are the most common block ciphers in use today. From the student's point of view, DES is still interesting to study, but due to its small 56-bit key size, it is considered insecure. In 1999, two partners, Electronic Frontier Foundation and distributed.net collaborated to publicly break a DES key in 22 hours and 15 minutes. Here, we will use the PyCrypto library to demonstrate how to use DES to encrypt a message. The following code is using the ECB block mode; you will learn about the various modes later in this chapter. To execute the following recipe, perform a pip install PyCrypto:

```
from Crypto.Cipher import DES
key = b'shhhhhh!'
origText = b'The US Navy has submarines in Kingsbay!!'
des = DES.new(key, DES.MODE_ECB)
ciphertext = des.encrypt(origText)
plaintext = des.decrypt(ciphertext)
print('The original text is {}'.format(origText))
print('The ciphertext is {}'.format(ciphertext))
print('The plaintext is {}'.format(plaintext))
print()
```

This should produce the following output:

```
The original text is b'The US Navy has submarines in Kingsbay!!'
The ciphertext is b'\xf6\x0bb\xf9L\x15I\xf9\x0f\xe2\xee_^\xdaQX\xe1y\
xe5\xea\xd3Z\xc8y\xee\xd3\x86H\xf0Nn\x83\x93\nOd@6H\xd4'
The plaintext is b'The US Navy has submarines in Kingsbay!!'
Press any key to continue . . .
```

The key was 'shhhhhh!' and the message was 'The US Navy has submarines in Kingsbay!!'. The ciphertext was 40 bytes long; 40 mod 8 = 0, so there is no need to pad this example. If you were to implement a block cipher in reality, you should use a padding function that ensures the block length.

What is DES?

Data Encryption Standard (DES) is a block cipher with a 56-bit key length that has played a significant role in data security**.** Data encryption standard (DES) has been found vulnerable to very powerful attacks therefore, the popularity of DES has been found slightly on the decline. DES is a block cipher and encrypts data in blocks of size of **64 bits** each, which means 64 bits of plain text go as the input to DES, which produces 64 bits of ciphertext. The same algorithm and key are used for encryption and decryption, with minor differences. The key length is **56 bits**.

The basic idea is shown below:

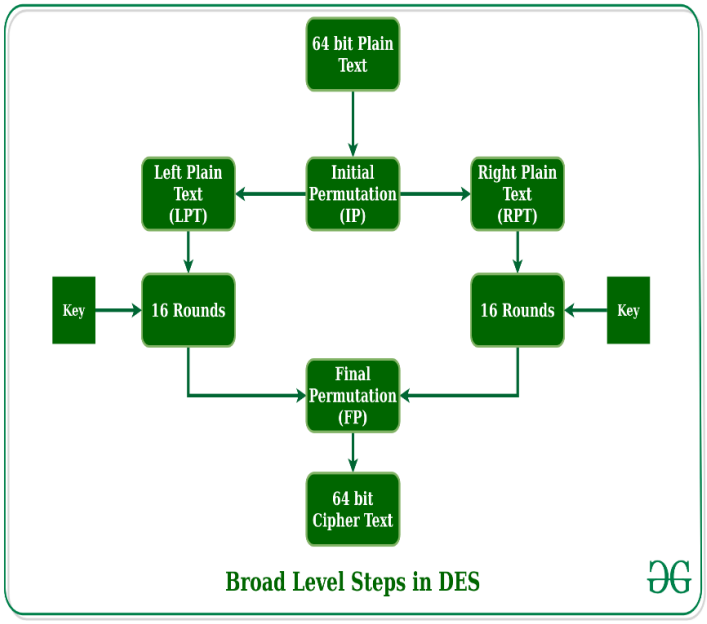
We have mentioned that DES uses a 56-bit key. Actually, The initial key consists of 64 bits. However, before the DES process even starts, every 8th bit of the key is discarded to produce a 56-bit key. That is bit positions 8, 16, 24, 32, 40, 48, 56, and 64 are discarded.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64

Figure - discording of every 8th bit of original key

Thus, the discarding of every 8th bit of the key produces a **56-bit key** from the original **64-bit key**. DES is based on the two fundamental attributes of cryptography: substitution (also called confusion) and transposition (also called diffusion). DES consists of 16 steps, each of which is called a round. Each round performs the steps of substitution and transposition. Let us now discuss the broad-level steps in DES.

- In the first step, the 64-bit plain text block is handed over to an initial Permutation (IP) function.
  - The initial permutation is performed on plain text.
  - Next, the initial permutation (IP) produces two halves of the permuted block; saying Left Plain Text (LPT) and Right Plain Text (RPT).
  - Now each LPT and RPT go through 16 rounds of the encryption process.
  - In the end, LPT and RPT are rejoined and a Final Permutation (FP) is performed on the combined block
  - The result of this process produces 64-bit ciphertext.



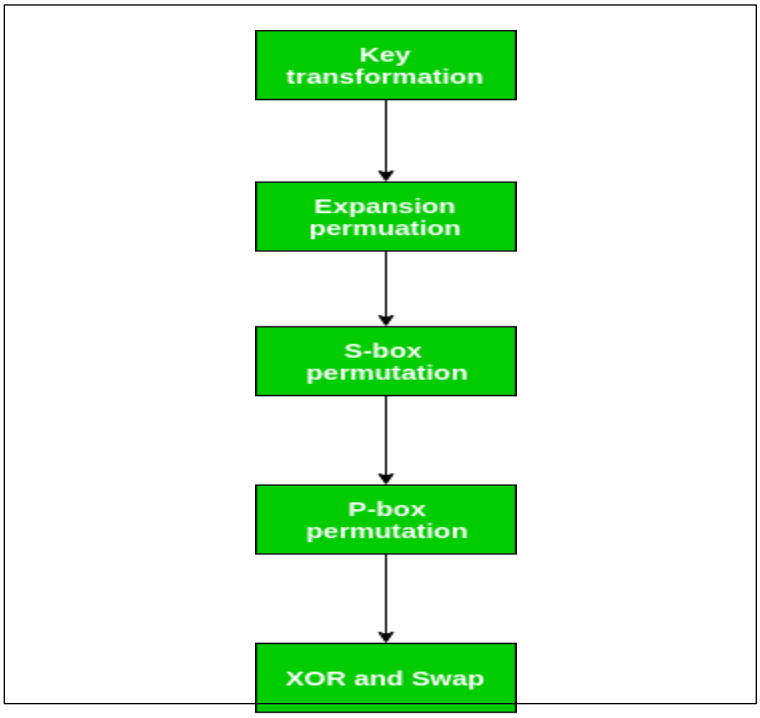
Initial Permutation (IP)

As we have noted, the initial permutation (IP) happens only once and it happens before the first round. It suggests how the transposition in IP should proceed, as shown in the figure. For example, it says that the IP replaces the first bit of the original plain text block with the 58th bit of the original plain text, the second bit with the 50th bit of the original plain text block, and so on. This is nothing but jugglery of bit positions of the original plain text block. The same rule applies to all the other bit positions shown in the figure.

58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6	64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3
61	33	45	37	29	21	13	5	63	55	47	39	31	23	15	7

Figure - Initial permutation table

As we have noted after IP is done, the resulting 64-bit permuted text block is divided into two half blocks. Each half-block consists of 32 bits, and each of the 16 rounds, in turn, consists of the broad-level steps outlined in the figure.



Step 1: Key transformation

We have noted initial 64-bit key is transformed into a 56-bit key by discarding every 8th bit of the initial key. Thus, for each a 56-bit key is available. From this 56-bit key, a different 48-bit Sub Key is generated during each round using a process called key transformation. For this, the 56-bit key is divided into two halves, each of 28 bits. These halves are circularly shifted left by one or two positions, depending on the round.

For example: if the round numbers 1, 2, 9, or 16 the shift is done by only one position for other rounds, the circular shift is done by two positions. The number of key bits shifted per round is shown in the figure.

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
#key bits shifted	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Figure - number of key bits shifted per round

After an appropriate shift, 48 of the 56 bits are selected. From the 48 we might obtain 64 or 56 bits based on requirement which helps us to recognize that this model is very versatile and can handle any range of requirements needed or provided. for selecting 48 of the 56 bits the table is shown in the figure given below. For instance, after the shift, bit number 14 moves to the first position, bit number 17 moves to the second position, and so on. If we observe the table , we will realize that it contains only 48-bit positions. Bit number 18 is discarded (we will not find it in the table), like 7 others, to reduce a 56-bit key to a 48-bit key. Since the key transformation process involves permutation as well as a selection of a 48-bit subset of the original 56-bit key it is called Compression Permutation.

14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

Figure - compression permutation

Because of this compression permutation technique, a different subset of key bits is used in each round. That makes DES not easy to crack.

Step 2: Expansion Permutation

Recall that after the initial permutation, we had two 32-bit plain text areas called Left Plain Text(LPT) and Right Plain Text(RPT). During the expansion permutation, the RPT is expanded from 32 bits to 48 bits. Bits are permuted as well hence called expansion permutation. This happens as the 32-bit RPT is divided into 8 blocks, with each block consisting of 4 bits. Then, each 4-bit block of the previous step is then expanded to a corresponding 6-bit block, i.e., per 4-bit block, 2 more bits are added.

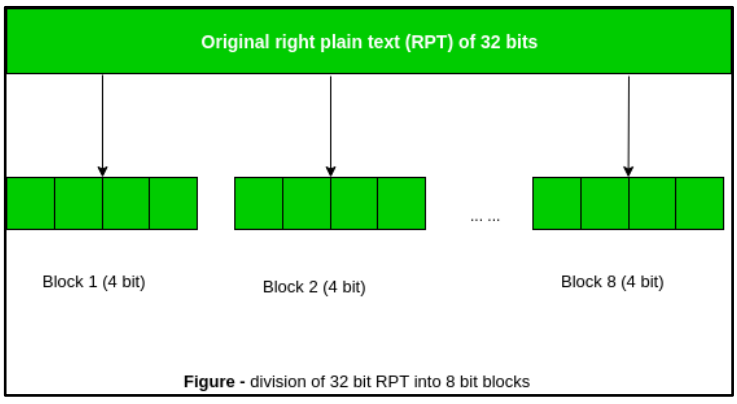


Figure - division of 32 bit RPT into 8 bit blocks

This process results in expansion as well as a permutation of the input bit while creating output. The key transformation process compresses the 56-bit key to 48 bits. Then the expansion permutation process expands the **32-bit RPT** to **48-bits**. Now the 48-bit key is XOR with 48-bit RPT and the resulting output is given to the next step, which is the **S-Box substitution**.

Python3 code for the above approach

```
# Hexadecimal to binary conversion

def hex2bin(s):
	mp = {'0': "0000",
		'1': "0001",
		'2': "0010",
		'3': "0011",
		'4': "0100",
		'5': "0101",
		'6': "0110",
		'7': "0111",
		'8': "1000",
		'9': "1001",
		'A': "1010",
		'B': "1011",
		'C': "1100",
		'D': "1101",
		'E': "1110",
		'F': "1111"}
	bin = ""
	for i in range(len(s)):
		bin = bin + mp[s[i]]
	return bin

# Binary to hexadecimal conversion


def bin2hex(s):
	mp = {"0000": '0',
		"0001": '1',
		"0010": '2',
		"0011": '3',
		"0100": '4',
		"0101": '5',
		"0110": '6',
		"0111": '7',
		"1000": '8',
		"1001": '9',
		"1010": 'A',
		"1011": 'B',
		"1100": 'C',
		"1101": 'D',
		"1110": 'E',
		"1111": 'F'}
	hex = ""
	for i in range(0, len(s), 4):
		ch = ""
```

```
                ch = ch + s[i]
                ch = ch + s[i + 1]
                ch = ch + s[i + 2]
                ch = ch + s[i + 3]
                hex = hex + mp[ch]

        return hex

# Binary to decimal conversion


def bin2dec(binary):

        binary1 = binary
        decimal, i, n = 0, 0, 0
        while(binary != 0):
                dec = binary % 10
                decimal = decimal + dec * pow(2, i)
                binary = binary//10
                i += 1
        return decimal

# Decimal to binary conversion


def dec2bin(num):
        res = bin(num).replace("0b", "")
        if(len(res) % 4 != 0):
                div = len(res) / 4
                div = int(div)
                counter = (4 * (div + 1)) - len(res)
                for i in range(0, counter):
                        res = '0' + res
        return res

# Permute function to rearrange the bits


def permute(k, arr, n):
        permutation = ""
        for i in range(0, n):
                permutation = permutation + k[arr[i] - 1]
        return permutation

# shifting the bits towards left by nth shifts


def shift_left(k, nth_shifts):
        s = ""
        for i in range(nth_shifts):
                for j in range(1, len(k)):
```

```
                        s = s + k[j]
                s = s + k[0]
                k = s
                s = ""
        return k

# calculating xow of two strings of binary number a and b


def xor(a, b):
        ans = ""
        for i in range(len(a)):
                if a[i] == b[i]:
                        ans = ans + "0"
                else:
                        ans = ans + "1"
        return ans


# Table of Position of 64 bits at initial level: Initial Permutation Table
initial_perm = [58, 50, 42, 34, 26, 18, 10, 2,
                                60, 52, 44, 36, 28, 20, 12, 4,
                                62, 54, 46, 38, 30, 22, 14, 6,
                                64, 56, 48, 40, 32, 24, 16, 8,
                                57, 49, 41, 33, 25, 17, 9, 1,
                                59, 51, 43, 35, 27, 19, 11, 3,
                                61, 53, 45, 37, 29, 21, 13, 5,
                                63, 55, 47, 39, 31, 23, 15, 7]

# Expansion D-box Table
exp_d = [32, 1, 2, 3, 4, 5, 4, 5,
                6, 7, 8, 9, 8, 9, 10, 11,
                12, 13, 12, 13, 14, 15, 16, 17,
                16, 17, 18, 19, 20, 21, 20, 21,
                22, 23, 24, 25, 24, 25, 26, 27,
                28, 29, 28, 29, 30, 31, 32, 1]

# Straight Permutation Table
per = [16, 7, 20, 21,
        29, 12, 28, 17,
        1, 15, 23, 26,
        5, 18, 31, 10,
        2, 8, 24, 14,
        32, 27, 3, 9,
        19, 13, 30, 6,
        22, 11, 4, 25]

# S-box Table
sbox = [[[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
                [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
                [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
```

```
                [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]],

                [[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],
                [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],
                [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],
                [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]],

                [[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],
                [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
                [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],
                [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]],

                [[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],
                [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],
                [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],
                [3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]],

                [[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
                [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
                [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],
                [11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]],

                [[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],
                [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],
                [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],
                [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]],

                [[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],
                [13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],
                [1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],
                [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]],

                [[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
                [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
                [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
                [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]]]

# Final Permutation Table
final_perm = [40, 8, 48, 16, 56, 24, 64, 32,
                        39, 7, 47, 15, 55, 23, 63, 31,
                        38, 6, 46, 14, 54, 22, 62, 30,
                        37, 5, 45, 13, 53, 21, 61, 29,
                        36, 4, 44, 12, 52, 20, 60, 28,
                        35, 3, 43, 11, 51, 19, 59, 27,
                        34, 2, 42, 10, 50, 18, 58, 26,
                        33, 1, 41, 9, 49, 17, 57, 25]


def encrypt(pt, rkb, rk):
        pt = hex2bin(pt)
```

```
	# Initial Permutation
	pt = permute(pt, initial_perm, 64)
	print("After initial permutation", bin2hex(pt))

	# Splitting
	left = pt[0:32]
	right = pt[32:64]
	for i in range(0, 16):
		# Expansion D-box: Expanding the 32 bits data into 48 bits
		right_expanded = permute(right, exp_d, 48)

		# XOR RoundKey[i] and right_expanded
		xor_x = xor(right_expanded, rkb[i])

		# S-boxex: substituting the value from s-box table by calculating row and column
		sbox_str = ""
		for j in range(0, 8):
			row = bin2dec(int(xor_x[j * 6] + xor_x[j * 6 + 5]))
			col = bin2dec(
				int(xor_x[j * 6 + 1] + xor_x[j * 6 + 2] + xor_x[j * 6 + 3] + xor_x[j * 6 +
4]))
			val = sbox[j][row][col]
			sbox_str = sbox_str + dec2bin(val)

		# Straight D-box: After substituting rearranging the bits
		sbox_str = permute(sbox_str, per, 32)

		# XOR left and sbox_str
		result = xor(left, sbox_str)
		left = result

		# Swapper
		if(i != 15):
			left, right = right, left
		print("Round ", i + 1, " ", bin2hex(left),
			" ", bin2hex(right), " ", rk[i])

	# Combination
	combine = left + right

	# Final permutation: final rearranging of bits to get cipher text
	cipher_text = permute(combine, final_perm, 64)
	return cipher_text


pt = "123456ABCD132536"
key = "AABB09182736CCDD"

# Key generation
# --hex to binary
key = hex2bin(key)
```

```
# --parity bit drop table
keyp = [57, 49, 41, 33, 25, 17, 9,
                    1, 58, 50, 42, 34, 26, 18,
                    10, 2, 59, 51, 43, 35, 27,
                    19, 11, 3, 60, 52, 44, 36,
                    63, 55, 47, 39, 31, 23, 15,
                    7, 62, 54, 46, 38, 30, 22,
                    14, 6, 61, 53, 45, 37, 29,
                    21, 13, 5, 28, 20, 12, 4]

# getting 56 bit key from 64 bit using the parity bits
key = permute(key, keyp, 56)

# Number of bit shifts
shift_table = [1, 1, 2, 2,
                            2, 2, 2, 2,
                            1, 2, 2, 2,
                            2, 2, 2, 1]

# Key- Compression Table : Compression of key from 56 bits to 48 bits
key_comp = [14, 17, 11, 24, 1, 5,
                            3, 28, 15, 6, 21, 10,
                            23, 19, 12, 4, 26, 8,
                            16, 7, 27, 20, 13, 2,
                            41, 52, 31, 37, 47, 55,
                            30, 40, 51, 45, 33, 48,
                            44, 49, 39, 56, 34, 53,
                            46, 42, 50, 36, 29, 32]

# Splitting
left = key[0:28] # rkb for RoundKeys in binary
right = key[28:56] # rk for RoundKeys in hexadecimal

rkb = []
rk = []
for i in range(0, 16):
        # Shifting the bits by nth shifts by checking from shift table
        left = shift_left(left, shift_table[i])
        right = shift_left(right, shift_table[i])

        # Combination of left and right string
        combine_str = left + right

        # Compression of key from 56 to 48 bits
        round_key = permute(combine_str, key_comp, 48)

        rkb.append(round_key)
        rk.append(bin2hex(round_key))

print("Encryption")
```

```
cipher_text = bin2hex(encrypt(pt, rkb, rk))
print("Cipher Text : ", cipher_text)

print("Decryption")
rkb_rev = rkb[::-1]
rk_rev = rk[::-1]
text = bin2hex(encrypt(cipher_text, rkb_rev, rk_rev))
print("Plain Text : ", text)
```

Output:

```
...60AF7CA5
Round 12 FF3C485F 22A5963B C2C1E96A4BF3
Round 13 22A5963B 387CCDAA 99C31397C91F
Round 14 387CCDAA BD2DD2AB 251B8BC717D0
Round 15 BD2DD2AB CF26B472 3330C5D9A36D
Round 16 19BA9212 CF26B472 181C5D75C66D

Cipher Text: C0B7A8D05F3A829C

Decryption

After initial permutation: 19BA9212CF26B472
After splitting: L0=19BA9212 R0=CF26B472

Round 1 CF26B472 BD2DD2AB 181C5D75C66D
Round 2 BD2DD2AB 387CCDAA 3330C5D9A36D
Round 3 387CCDAA 22A5963B 251B8BC717D0
Round 4 22A5963B FF3C485F 99C31397C91F
Round 5 FF3C485F 6CA6CB20 C2C1E96A4BF3
Round 6 6CA6CB20 10AF9D37 6D5560AF7CA5
Round 7 10AF9D37 308BEE97 02765708B5BF
Round 8 308BEE97 A9FC20A3 84BB4473DCCC
```

```
Round 9 A9FC20A3 2E8F9C65 34F822F0C66D

Round 10 2E8F9C65 A15A4B87 708AD2DDB3C0

Round 11 A15A4B87 236779C2 C1948E87475E

Round 12 236779C2 B8089591 69A629FEC913

Round 13 B8089591 4A1210F6 DA2D032B6EE3

Round 14 4A1210F6 5A78E394 06EDA4ACF5B5

Round 15 5A78E394 18CA18AD 4568581ABCCE

Round 16 14A7D678 18CA18AD 194CD072DE8C


Plain Text: 123456ABCD132536
```

Output:

```
Encryption:
After initial permutation: 14A7D67818CA18AD
After splitting: L0=14A7D678 R0=18CA18AD
Round 1 18CA18AD 5A78E394 194CD072DE8C
Round 2 5A78E394 4A1210F6 4568581ABCCE
Round 3 4A1210F6 B8089591 06EDA4ACF5B5
Round 4 B8089591 236779C2 DA2D032B6EE3
Round 5 236779C2 A15A4B87 69A629FEC913
Round 6 A15A4B87 2E8F9C65 C1948E87475E
Round 7 2E8F9C65 A9FC20A3 708AD2DDB3C0
Round 8 A9FC20A3 308BEE97 34F822F0C66D
Round 9 308BEE97 10AF9D37 84BB4473DCCC
Round 10 10AF9D37 6CA6CB20 02765708B5BF
Round 11 6CA6CB20 FF3C485F 6D5560AF7CA5
Round 12 FF3C485F 22A5963B C2C1E96A4BF3
Round 13 22A5963B 387CCDAA 99C31397C91F
Round 14 387CCDAA BD2DD2AB 251B8BC717D0
Round 15 BD2DD2AB CF26B472 3330C5D9A36D
Round 16 19BA9212 CF26B472 181C5D75C66D
Cipher Text: C0B7A8D05F3A829C
Decryption
After initial permutation: 19BA9212CF26B472
After splitting: L0=19BA9212 R0=CF26B472
Round 1 CF26B472 BD2DD2AB 181C5D75C66D
Round 2 BD2DD2AB 387CCDAA 3330C5D9A36D
Round 3 387CCDAA 22A5963B 251B8BC717D0
```

```
Round 4 22A5963B FF3C485F 99C31397C91F
Round 5 FF3C485F 6CA6CB20 C2C1E96A4BF3
Round 6 6CA6CB20 10AF9D37 6D5560AF7CA5
Round 7 10AF9D37 308BEE97 02765708B5BF
Round 8 308BEE97 A9FC20A3 84BB4473DCCC
Round 9 A9FC20A3 2E8F9C65 34F822F0C66D
Round 10 2E8F9C65 A15A4B87 708AD2DDB3C0
Round 11 A15A4B87 236779C2 C1948E87475E
Round 12 236779C2 B8089591 69A629FEC913
Round 13 B8089591 4A1210F6 DA2D032B6EE3
Round 14 4A1210F6 5A78E394 06EDA4ACF5B5
Round 15 5A78E394 18CA18AD 4568581ABCCE
Round 16 14A7D678 18CA18AD 194CD072DE8C
Plain Text: 123456ABCD132536
```

Graded Task 1

You have implemented DES there is built in implemented DES in python in crypto cipher module use it for encryption/decryption and provide output sample example

```
from Crypto.Cipher import DES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad

# DES key must be exactly 8 bytes long
key = get_random_bytes(8)

def des_encrypt(data, key):
    cipher = DES.new(key, DES.MODE_ECB)
    padded_data = pad(data, DES.block_size)
    encrypted_data = cipher.encrypt(padded_data)
    return encrypted_data

def des_decrypt(encrypted_data, key):
    cipher = DES.new(key, DES.MODE_ECB)
    decrypted_data = unpad(cipher.decrypt(encrypted_data), DES.block_size)
    return decrypted_data

# Example usage
if __name__ == "__main__":
```

```
# Input data (must be bytes)
data = b"Secret123"

print(f"Original Data: {data}")

# Encrypt the data
encrypted_data = des_encrypt(data, key)
print(f"Encrypted Data: {encrypted_data}")

# Decrypt the data
decrypted_data = des_decrypt(encrypted_data, key)
print(f"Decrypted Data: {decrypted_data}")
```

Graded Task 2

Visualization of MITM Attack Flow:

```
Attacker intercepts plaintext (P) and ciphertext (C)

1. Guess K1
   P --[Encrypt with K1]--> Intermediate Value 1 (I1)

2. Guess K2
   C --[Decrypt with K2]--> Intermediate Value 2 (I2)

3. If I1 == I2, then K1 and K2 are likely correct.
   Found the DES key: K = K1 + K2
```

Assumptions:

- The attacker knows or can guess some **plaintext-ciphertext pairs** (this is called a **known-plaintext attack**).
- The DES encryption and decryption processes can be divided into independent stages, allowing the attacker to perform partial encryption and decryption.

Steps in Meet-in-the-Middle Attack:

1. **Intercept the Ciphertext**: The attacker intercepts a **known plaintext** (i.e., a piece of the original message that is known or can be guessed) and the corresponding **ciphertext** (i.e., the encrypted version of that message).
2. **Divide the DES Algorithm into Two Stages**:
    - DES operates in multiple rounds of encryption, but the MITM attack divides this into two stages:
        1. **First encryption stage** (Encrypt a known plaintext).
        2. **Second decryption stage** (Decrypt a known ciphertext).

    The attack leverages the fact that the encryption can be split into these stages, and the

intermediate value after one encryption round should match with the decrypted intermediate value after one decryption round.

3. **Guess the First Half of the Key**: The attacker guesses the first half of the key (let's call it `K1`). The attacker encrypts the known plaintext using `K1` and stores the result (intermediate encryption value).
4. **Guess the Second Half of the Key**: The attacker guesses the second half of the key (let's call it `K2`). The attacker decrypts the intercepted ciphertext using `K2` and stores the result (intermediate decryption value).
5. **Matching Intermediate Values**:
    - The attacker compares the intermediate values from the first stage (encrypting with `K1`) and the second stage (decrypting with `K2`).
    - If the intermediate values match, the attacker has likely found the correct combination of the two keys (`K1` and `K2`), which together form the full DES key.
    - Since DES uses a single 56-bit key, this is broken down into halves for this type of attack.

Home –Task: Advanced Encryption Standard (AES)

AES stands for Advanced Encryption Standard, and it is the only public encryption scheme that the NSA approves for confidential information. We focus on its use as our main block cipher from now on. AES is the current de facto block cipher, and it works on 16 bytes at a time. It has three possible key lengths: 16-byte, 24-byte, or 32-byte. We know that a block cipher is effectively a deterministic permutation on binary strings, like a fixed-length reversible hash. Given a proper-length key and a 16-byte input we should always get the same 16-byte output. Note that there are typically three ways to work with bytes: plain ASCII, hex digest, and base64 (we haven't played with this yet but we will). A good chunk of your bugs come from transferring between hex and raw. You explore AES in the next chapter as you manipulate images.

Using AES with Python Earlier in this chapter, you were introduced to PyCrypto as a Python module that enables block ciphers using DES; it also has methods for encrypting AES. The PyCrypto module is similar to the Java Cryptography Extension (JCE) that is used in Java. The first step we will take in our AES encryption is to generate a strong key. As you know, the stronger the key, the stronger the encryption. The key we use for our encryption is oftentimes the weakest link in our encryption chain. The key we select should not be guessable and should provide sufficient entropy, which simply means that the key should lack order or predictability. The following Python code will create a random key that is 16 bytes:

```
import os
import binascii
key = binascii.hexlify(os.urandom(16))
print ('key', [x for x in key] )
key [97, 53, 99, 97, 102, 99, 102, 102, 50, 101, 98, 57, 97, 51, 50, 50,
51, 52, 102, 49, 101, 51, 102, 52, 100, 49, 48, 51, 51, 49, 56, 51]
```

Now that you have generated a key, you will need an initialization vector. The IV should be generated for each message to ensure a different encrypted text each time the message is encrypted. The IV adds significant protection in case the message is intercepted; it should mitigate the use of cryptanalysis to infer message or key data. The IV is required to be transmitted to the message receiver to ensure proper

decryption, but unlike the message key, the IV does not need to be kept secret. You can add the IV to process the encrypted text. The message receiver will need to know where the IV is located inside the mes sage. You can create a random IV by using the following snippet; note the use of random.randint. This method of generating random numbers is less effective and it has a lower entropy, but in this case we are using it to create the IV that will be used in the encryption process so there is less concern with the use of randint here:

```
 iv = ''.join([chr(random.randint(0, 0xFF)) for i in range(16)]) )])
```

The next step in the process is to create the ciphertext. In this example, we will use the CBC mode; this links the current block to the previous block in the stream. See the previous section to review the various AES block modes.

Remember that for this implementation of AES using PyCrypto, you will need to ensure that you pad the block to guarantee you have enough data in the block:

```
aes = AES.new(key, AES.MODE_CBC, iv)
data = 'Playing with AES' # <- 16 bytes
encd = aes.encrypt(data)
```

To decrypt the ciphertext, you will need the key that was used for the encryption. Transporting the key, inside itself, can be a challenge. You will learn about key exchange in a later chapter. In addition to the key, you will also need the IV. The IV can be transmitted over any line of communication as there are no requirements to encrypt it. You can safely send the IV along with the encrypted file and embed it in plaintext, as shown here:

```
from base64 import b64encode
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
import binascii, os
import random
data = b"secret"
key = binascii.hexlify(os.urandom(16))
iv = ''.join(chr(random.randint(0, 0xFF)) for i in range(16))
#print ('key: ', [x for x in key] )
#print()
cipher = AES.new(key, AES.MODE_CBC)
ct_bytes = cipher.encrypt(pad(data, AES.block_size))
ct = b64encode(ct_bytes).decode('utf-8')
print('iv: {}'.format(iv))
print()
print('ciphertext: {}'.format(ct))
print()
iv: öhLÒô™Ï2q]>°mâ
ciphertext: BDE+z8ME6r0QgkraNXLuuQ==
```

File Encryption Using AES

Next, you can encrypt a file using AES by implementing the following Python recipe. The primary difference is opening the file and passing the packs into blocks:

```
aes = AES.new(key, AES.MODE_CBC, iv)
filesize = os.path.getsize(infile)
```

```
with open (encrypted, ‘w’) as fout:
    fout.write(struct.pack(‘<Q’, filesize))
fout.write(iv)
```

File Decryption Using AES

To decrypt the previous example, use the following code to reverse the process:

```
with open(verfile, 'w') as fout:
  while True:
      data = fin.read(sz)
      n = len(data)
      if n == 0:
          break
      decd = aes.decrypt(data)
      n = len(decd)
      if fsz > n:
          fout.write(decd)
      else:
          fout.write(decd[:fsz]) # <- remove padding on last block
      fsz -= n
```