CS553: Cryptography

Assignment 7: Solutions

Rohit Das (11910230)

November 11, 2019

1. Mix-Column Transitions

Python Code (Python3): AES.py

```
import numpy as np

sbox = [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]

r_con = [
    0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40,
    0x80, 0x1B, 0x36, 0x6C, 0xD8, 0xAB, 0x4D, 0x9A,
    0x2F, 0x5E, 0xBC, 0x63, 0xC6, 0x97, 0x35, 0x6A,
    0xD4, 0xB3, 0x7D, 0xFA, 0xEF, 0xC5, 0x91, 0x39,
]

keyset = []

def AESinput(msg):

    state = np.empty([4,4],dtype=object)
    msg = ["{:02x}".format(ord(m),'x') for m in msg]
    if len(msg) > 16:
        print("Message size greater than 16 bytes")
        exit(0)
```

```python
    i = 0
    # padding scheme: ANSI X9.23
    while len(msg) < 15:
        i += 1
        msg.append("{:02x}".format(0,'x'))
    msg.append("{:02x}".format(i + 1,'x'))

    l = 0
    for i in range(0,4):
        for j in range(0,4):
            state[j][i] = msg[l]
            l += 1

    # state[0][0] = int(msg[0],2)
    return state

def subBytes(state):
    for i in range(0,4):
        for j in range(0,4):
            state[j][i] = "{:02x}".format(sbox[int(state[j][i],16)],'x')
    return state

def shiftRows(state):
    # print(state)
    for i in range(0,4):
        state[i] = np.roll(state[i],-1 * i)
    return state

def galoisMult(a, b):
    prod = 0
    b7Set = 0
    for i in range(8):
        if b & 1 == 1:
            prod ^= a
        b7Set = a & 0x80
```

```
        a <<= 1
        if b7Set == 0x80:
            a ^= 0x1b
        b >>= 1
    return prod % 256

def mixColumns(state):
    state = np.transpose(state)
    temp = np.zeros([4,4],dtype=object)
    # print(state)
    for t,i in zip(temp,range(4)):
        t[0] = "{:02x}".format(galoisMult(int(state[i][0],16),2) ^
            ↪ galoisMult(int(state[i][1],16),3) ^ galoisMult(int(state[i
            ↪ ][2],16),1) ^ galoisMult(int(state[i][3],16),1),'x')
        t[1] = "{:02x}".format(galoisMult(int(state[i][0],16),1) ^
            ↪ galoisMult(int(state[i][1],16),2) ^ galoisMult(int(state[i
            ↪ ][2],16),3) ^ galoisMult(int(state[i][3],16),1),'x')
        t[2] = "{:02x}".format(galoisMult(int(state[i][0],16),1) ^
            ↪ galoisMult(int(state[i][1],16),1) ^ galoisMult(int(state[i
            ↪ ][2],16),2) ^ galoisMult(int(state[i][3],16),3),'x')
        t[3] = "{:02x}".format(galoisMult(int(state[i][0],16),3) ^
            ↪ galoisMult(int(state[i][1],16),1) ^ galoisMult(int(state[i
            ↪ ][2],16),1) ^ galoisMult(int(state[i][3],16),2),'x')

    return np.transpose(temp)

def keySchedule(seed,rounds):
    keyset = []
    if len(seed) > 16:
        seed = seed[0:16]
    keyState = AESinput(seed)
    # print(keyState)
    temp = np.zeros([4,4],dtype=object)

    for n in range(rounds + 1):
```

```python
        # print("Iteration:",n)
        # print("keyState:",keyState)
        temp = keyState.copy()
        a = temp[0][3]
        for i in range(3):
            temp[i][3] = temp[i + 1][3]
        temp[3][3] = a
        # print("temp:",temp)

        temp = subBytes(temp)
        temp[0][3] = "{:02x}".format(int(temp[0][3],16) ^ r_con[n],'x')
        # print("temp:",temp)
        # print("keyState:",keyState)

        for i in range(4):
            temp[i][0] = "{:02x}".format(int(keyState[i][0],16) ^ int(
                ↪ temp[i][3],16),'x')
        # print("temp:",temp)

        for j in range(1,4):
            for i in range(4):
                temp[i][j] = "{:02x}".format(int(keyState[i][j],16) ^
                    ↪ int(temp[i][j - 1],16),'x')
        keyState = temp
        # print("keyState:",keyState)
        keyset.append(keyState)

    return keyset

def addKey(state,key):
    for i in range(4):
        for j in range(4):
            state[i][j] = "{:02x}".format(int(state[i][j],16) ^ int(key[
                ↪ i][j],16),'x')
    return state
```

```
# print("{:08b}".format(galoisMult(0xff,3),'b'))
# print(AESinput('Rohit'))
# print(mixColumns(shiftRows(AESinput('Rohit'))))
# print(AESinput('0000000000000000'))

def AES(msg,rounds,key):
    msg = AESinput(msg)
    print("Msg:")
    print(msg)
    keyset = keySchedule(key,rounds)
    addKey(msg,keyset[0])

    for n in range(1,rounds):
        msg = mixColumns(shiftRows(subBytes(msg)))
        msg = addKey(msg,keyset[n])

    msg = shiftRows(subBytes(msg))
    msg = addKey(msg, keyset[rounds])

    return msg

# print(AES('Rohit',10,'1234567800000000'))
```

2. Integral Distinguisher

Python Code (Python3): integral_crypt.py

```
import numpy as np
import AES # imports from AES.py

def makeIntegral(r,c): # creates an integral 3D list
    integral_state = np.arange(16*256,dtype=object).reshape(256,4,4)
    random = np.random.randint(0,256,16)

    for i in range(256):
        l = 0
        for j in range(4):
            for k in range(4):
                if j == r + 1 and k == c + 1:
                    integral_state[i][j - 1][k - 1] = "{:02x}".format(i,
                      ↪ 'x')
# filling other bytes with random constants, same for all 256 states
                else:
                    integral_state[i][j - 1][k - 1] = "{:02x}".format(
                      ↪ random[l], 'x')
                    l += 1
    return integral_state

def isAll(a): # prints 2D list with All bytes marked
    isAllList = np.empty([4,4],dtype=str)
    for j in range(4):
        for k in range(4):
            isAllFlag = True
            n = [i for i in range(256)]
            for i in range(256):
                if int(a[i][j][k],16) not in n:
                    isAllFlag = False
                else:
                    n[int(a[i][j][k],16)] = -1
```

```
                    isAllFlag = True

            if isAllFlag == True:
                isAllList[j][k] = 'A'
            else:
                isAllList[j][k] = ''

    return isAllList

def isConst(a): # prints 2D list with Constant bytes marked
    isConstList = np.empty([4,4],dtype=str)
    for j in range(4):
        for k in range(4):
            isConstFlag = True
            for i in range(256):
                # checks if all bytes are same
                if a[i][j][k] != a[0][j][k]:
                    isConstFlag = False
                else:
                    isConstFlag = True

            if isConstFlag == True:
                isConstList[j][k] = 'C'
            else:
                isConstList[j][k] = ''

    return isConstList

def isBalanced(a): # prints 2D list with Balanced bytes marked
    isBalancedList = np.empty([4,4],dtype=str)
    for j in range(4):
        for k in range(4):
            isBalancedFlag = 0
            for i in range(256):
                isBalancedFlag = isBalancedFlag ^ int(a[i][j][k],16)
```

```
            if isBalancedFlag == 0:
                isBalancedList[j][k] = 'B'
            else:
                isBalancedList[j][k] = ''

    return isBalancedList

r = int(input("Enter row for All byte: "),10)
c = int(input("Enter column for All byte: "),10)

integral_state = makeIntegral(r,c) # choosing which byte to make All
seed = input("Enter key:") # choosing key for key schedule
keyset = AES.keySchedule(seed, 3)
AES_func = [AES.subBytes, AES.shiftRows, AES.mixColumns, AES.addKey]

print("Round 0 AddKey:")
for i in integral_state:
    i = AES_func[3](i, keyset[0])
print(np.core.defchararray.add(isConst(integral_state),isAll(integral_state)))

# 3-round integral distinguisher
for j in range(3):
    print("Round",j + 1,":")
    for k in range(4):
        for i in range(len(integral_state)):
            # print("Round:", i)
            if k == 3:
                integral_state[i] = AES_func[k](integral_state[i], keyset[j + 1])
            else:
                integral_state[i] = AES_func[k](integral_state[i])

        print("After",AES_func[k].__name__,":")
```

```
        temp = np.core.defchararray.add(isConst(integral_state),isAll(
            ↪ integral_state))

        if '' in temp:
            print(isBalanced(integral_state))
        else:
            print(temp)
```

Output: Row: 1, Col: 1, Key: 1234

```
Round 0 AddKey:
[['C' 'C' 'C' 'C']
 ['C' 'A' 'C' 'C']
 ['C' 'C' 'C' 'C']
 ['C' 'C' 'C' 'C']]
Round 1 :
After subBytes :
[['C' 'C' 'C' 'C']
 ['C' 'A' 'C' 'C']
 ['C' 'C' 'C' 'C']
 ['C' 'C' 'C' 'C']]
After shiftRows :
[['C' 'C' 'C' 'C']
 ['A' 'C' 'C' 'C']
 ['C' 'C' 'C' 'C']
 ['C' 'C' 'C' 'C']]
After mixColumns :
[['A' 'C' 'C' 'C']
 ['A' 'C' 'C' 'C']
 ['A' 'C' 'C' 'C']
 ['A' 'C' 'C' 'C']]
After addKey :
[['A' 'C' 'C' 'C']
 ['A' 'C' 'C' 'C']
 ['A' 'C' 'C' 'C']
 ['A' 'C' 'C' 'C']]
```

```
Round 2 :
After subBytes :
[['A' 'C' 'C' 'C']
 ['A' 'C' 'C' 'C']
 ['A' 'C' 'C' 'C']
 ['A' 'C' 'C' 'C']]
After shiftRows :
[['A' 'C' 'C' 'C']
 ['C' 'C' 'C' 'A']
 ['C' 'C' 'A' 'C']
 ['C' 'A' 'C' 'C']]
After mixColumns :
[['A' 'A' 'A' 'A']
 ['A' 'A' 'A' 'A']
 ['A' 'A' 'A' 'A']
 ['A' 'A' 'A' 'A']]
After addKey :
[['A' 'A' 'A' 'A']
 ['A' 'A' 'A' 'A']
 ['A' 'A' 'A' 'A']
 ['A' 'A' 'A' 'A']]
Round 3 :
After subBytes :
[['A' 'A' 'A' 'A']
 ['A' 'A' 'A' 'A']
 ['A' 'A' 'A' 'A']
 ['A' 'A' 'A' 'A']]
After shiftRows :
[['A' 'A' 'A' 'A']
 ['A' 'A' 'A' 'A']
 ['A' 'A' 'A' 'A']
 ['A' 'A' 'A' 'A']]
After mixColumns :
[['B' 'B' 'B' 'B']
 ['B' 'B' 'B' 'B']
```

```
 ['B' 'B' 'B' 'B']
 ['B' 'B' 'B' 'B']]
After addKey :
[['B' 'B' 'B' 'B']
 ['B' 'B' 'B' 'B']
 ['B' 'B' 'B' 'B']
 ['B' 'B' 'B' 'B']]
```

3. Fault Tolerance: Sypher004

Python Code (Python3): Sypher004.py

```
import numpy as np


sbox1 ={0x0: 0x6, 0x1: 0x4, 0x2: 0xc, 0x3: 0x5, 0x4: 0x0,
        0x5: 0x7, 0x6: 0x2, 0x7: 0xe, 0x8: 0x1, 0x9: 0xf,
        0xa: 0x3, 0xb: 0xd, 0xc: 0x8, 0xd: 0xa, 0xe: 0x9,
        0xf: 0xb}


def sbox_4x(msg, bits): # 4-input sbox
    if len(msg) != bits:# check for message length
        exit("Input size should be of " + format(4*bits) + " bits")
    subs = []
    for m in msg: # check for invalid literal
        if m not in [format(i,'x') for i in sbox1]:
            exit("Invalid literal")
        subs.append(format(sbox1[int(m,16)],'x'))
    return subs


def sbox_4x_inv(cipher, bits):
    if len(cipher) != bits:# check for message length
        exit("Input size should be of " + format(4*bits) + " bits")
    subs = []
    for c in cipher:
        if c not in [format(i,'x') for i in sbox1.values()]:
            exit("Invalid literal")
        for _c,m in sbox1.items():
            if c == format(m,'x'):
                subs.append(format(_c,'x'))
    return subs


def pbox(sbox_out,bits):
    if len(sbox_out) != bits:
        exit("Too small sbox output!!")
```

```
    perm = [b for a in sbox_out # changing hex to binary
            for b in list("{0:04b}".format(int(a,16)))]
    # pbox = sbox output in numpy array and transposing
    perm = np.asarray(perm)
    perm = np.reshape(perm,(4,4)).transpose()
    return "".join([format(int("".join(a),2),'x')
                    for a in perm])

def sypher004(msg, bits):
    if len(msg) != bits:# check for message length
        exit("Plaintext size should be of " + format(4*bits) + " bits")

    keys = ['340b','3ffc','edfd','8c7d','0696','ffff']
    msg = "{:04x}".format(int(msg,16) ^ int(keys[0],16),'x')

    for i in range(1,len(keys) - 1):
        msg = "".join(sbox_4x(msg,bits))
        msg = pbox(msg,bits)

        msg = "{:04x}".format(int(msg,16)^int(keys[i],16),'x')

    msg = "".join(sbox_4x(msg,bits))
    msg = "{:04x}".format(int(msg,16)^int(keys[5],16),'x')

    return msg

def sypher004_inv(cipher, bits): # decryption of Sypher004
    if len(cipher) != bits:# check for message length
        exit("Ciphertext size should be of " + format(4*bits) + " bits")

    keys = ['340b','3ffc','edfd','8c7d','0696','ffff']
    cipher = "{:04x}".format(int(cipher,16) ^ int(keys[len(keys) -
        ↪ 1],16)
                                ,'x')
    cipher = "".join(sbox_4x_inv(cipher,bits))
```

```
    for i in range(len(keys) - 2,0,-1):
        cipher = "{:04x}".format(int(cipher,16)^int(keys[i],16),'x')
        cipher = pbox(cipher,bits)
        cipher = "".join(sbox_4x_inv(cipher,bits))

    cipher = "{:04x}".format(int(cipher,16)^int(keys[0],16),'x')

    return cipher

def cfb_enc(msg,IV,t,sypher004):
    x = IV
    cipher = []
    for m in msg:
        c = int(m,16) ^ int(bin(int(sypher004(x,4),16))[:t],2) # msb
            ↪ from x
        cipher.append(format(c,'x'))
        x = "{:04x}".format(((int(x,16) << t) | c) & 255,'x')
    return "".join(cipher)

def cfb_dec(cipher,IV,t,sypher004):
    x = IV
    msg = []
    for c in cipher:
        m = int(c,16) ^ int(bin(int(sypher004(x,4),16))[:t],2) # msb
            ↪ from x
        msg.append(format(m,'x'))
        x = "{:04x}".format(((int(x,16) << t) ^ int(c,16)) % 255,'x')
    return "".join(msg)

# For OFB, encryption is same as decryption, since keystream xor m = c.
def ofb_enc(msg,IV,t,sypher004):
    x = IV
    cipher = []
    for m in msg:
```

```
        x = sypher004(x,4)
        c = int(m,16) ^ int(bin(int(x,16))[:t],2) # msb from x
        cipher.append(format(c,'x'))
    return "".join(cipher)

def cipher_mode(IV,msg,mode_enc,mode_dec,mode,sypher004):
    c1 = mode_enc(msg,IV,4,sypher004)
    print("IV used:",IV)
    print("Encryption of msg:",msg,", using Sypher004 -",mode,":", c1)
    print("".join("{:04b}".format(int(c,16)) for c in c1))
    print("Flipping 6th bit from the right:")
    c1 = format(int(c1,16) ^ 32,'x')
    print("".join("{:04b}".format(int(c,16)) for c in c1))
    print("Decryption of modified cipher: ")
    c1 = mode_dec(c1,IV,4,sypher004)
    print("".join("{:04b}".format(int(c,16)) for c in c1),"<-",c1)
    print("".join("{:04b}".format(int(m,16)) for m in msg),"<- Original
      ↪ message")
    print()

IV = input("Enter IV: ")
if len(IV) < 4:
    exit("IV should be minimum of 16 bits")
elif len(IV) > 4:
    IV = IV[-4:]

msg = input("Enter message (multiple of 16 bits): ")
if len(msg) % 4 != 0:
    exit("Message length not a multiple of 16 bits!!")

print()
print("Using Sypher004 encryption:")
print("-" * 28)

print("Mode: CFB")
```

```
cipher_mode(IV,msg,cfb_enc,cfb_dec,'CFB',sypher004)

print("Mode: OFB")
cipher_mode(IV,msg,ofb_enc,ofb_enc,'OFB',sypher004)

print("Using Sypher004 decryption:")
print("-" * 28)

print("Mode: CFB")
cipher_mode(IV,msg,cfb_enc,cfb_dec,'CFB',sypher004_inv)

print("Mode: OFB")
cipher_mode(IV,msg,ofb_enc,ofb_enc,'OFB',sypher004_inv)
```

Output: IV: 1234. Message: 1234abcd.

```
Using Sypher004 encryption:
----------------------------
Mode: CFB
IV used: 1234
Encryption of msg: 1234abcd , using Sypher004 - CFB : 201689ff
00100000000101101000100111111111
Flipping 6th bit from the right:
00100000000101101000100111011111
Decryption of modified cipher:
00010010001101001010101011111100 <- 1234aafc
00010010001101001010101111001101 <- Original message

Mode: OFB
IV used: 1234
Encryption of msg: 1234abcd , using Sypher004 - OFB : 201699fe
00100000000101101001100111111110
Flipping 6th bit from the right:
00100000000101101001100111011110
Decryption of modified cipher:
00010010001101001010101111101101 <- 1234abed
```

```
00010010001101001010101111001101 <- Original message

Using Sypher004 decryption:
________________________________
Mode: CFB
IV used: 1234
Encryption of msg: 1234abcd , using Sypher004 - CFB : 300799ee
00110000000001111001100111101110
Flipping 6th bit from the right:
00110000000001111001100111001110
Decryption of modified cipher:
00010011001101001010101011111101 <- 1334aafd
00010010001101001010101111001101 <- Original message

Mode: OFB
IV used: 1234
Encryption of msg: 1234abcd , using Sypher004 - OFB : 301698fe
00110000000101101001100011111110
Flipping 6th bit from the right:
00110000000101101001100011011110
Decryption of modified cipher:
00010010001101001010101111101101 <- 1234abed
00010010001101001010101111001101 <- Original message
```

4. Modes

4..1 ECB Encryption

Bash Script File: enc_pic.sh

```
identify IITBhilaiLogo.png
convert -depth 32 IITBhilaiLogo.png IITBhilaiLogo.rgba
openssl aes-256-ecb -nosalt -in IITBhilaiLogo.rgba -out
IITBhilaiLogoEnc.rgba
pass: 'CS553'
convert -size 400x400 -depth 32 IITBhilaiLogoEnc.rgba
IITBhilaiLogoEnc.png
```

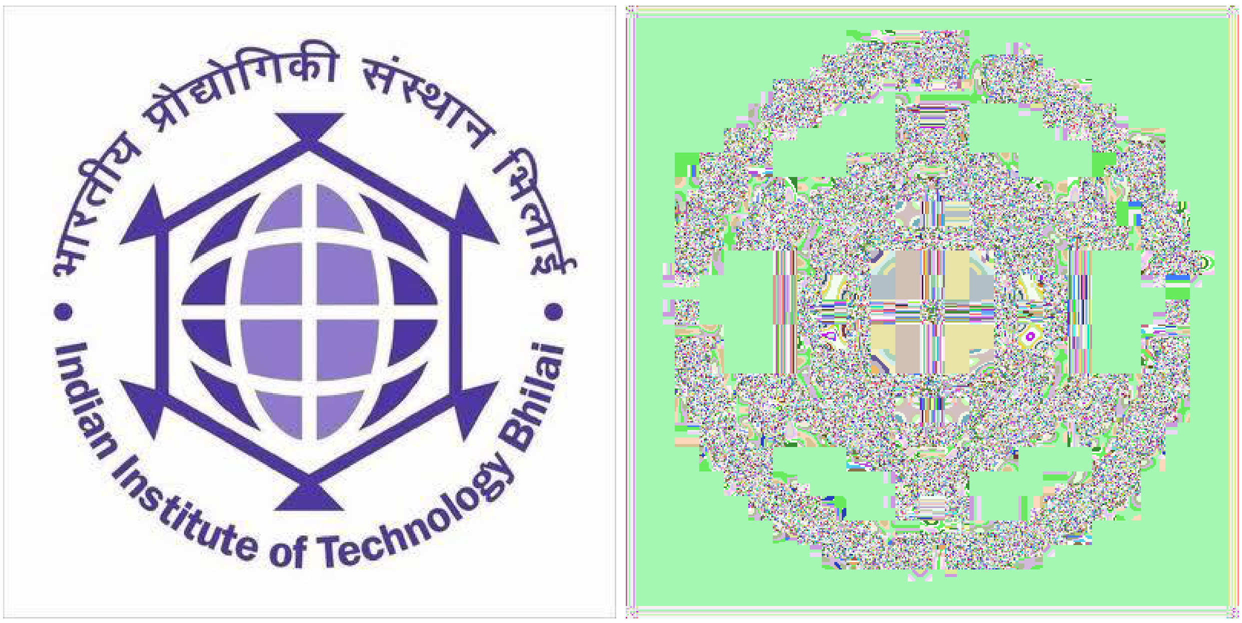

Figure 1: Left: Original IIT Bhilai Logo. Right: Encrypted using aes-256-cbc

4..2 Need for Pre-IV

In CBC and CFB modes of encryption, being able to predict the IV will lead to leaking the message itself. To make the IV unpredictable, a notion of Pre-IV is used. It can be a nonce generated from a counter, which is then encrypted to create the IV.

$$IV = ENC_k(\text{PRE-IV})$$

Other theoretical works suggest using some key k' derived from k that is used for encryption.

$$IV = ENC_{k'}(\text{PRE-IV})$$