# Practical Assignment 3,4,5

*Network Security CSE 537*

Submitted By:

**Ankur Sonawane**

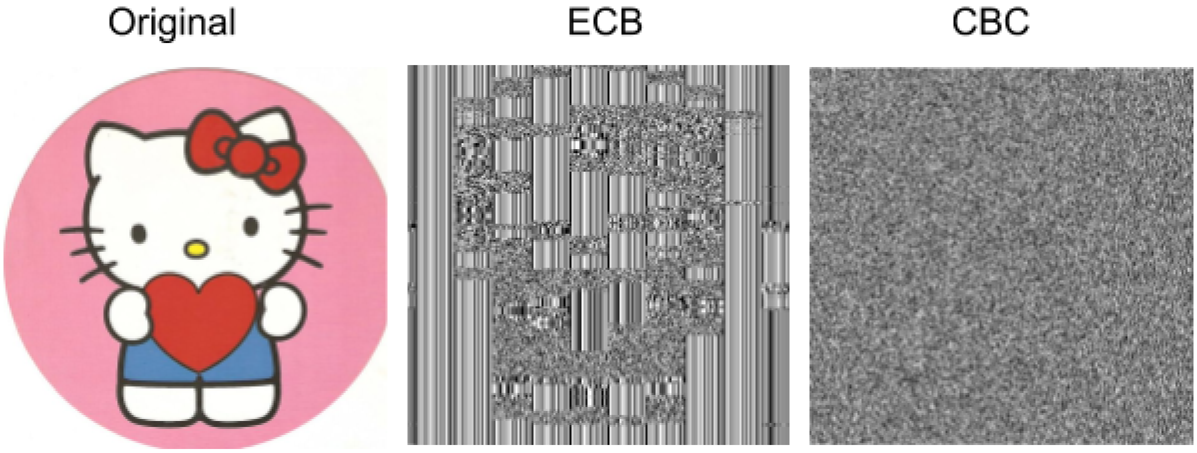
Electrical Engineering
16085073

Submitted To:

**Prof. KK Shukla**

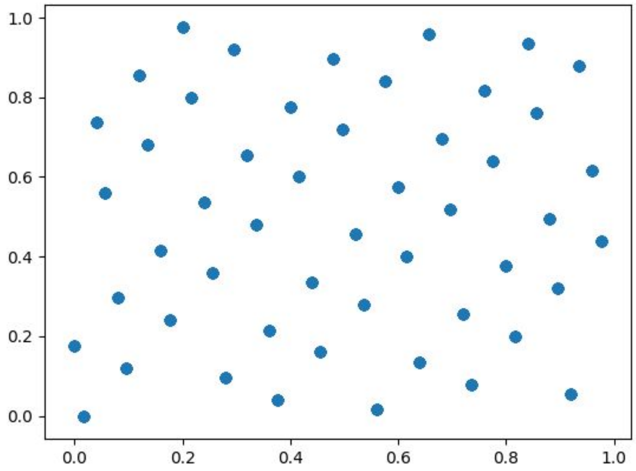
Department of Computer Science, IITBHU

## Practical Assignment 3: Using a binary image, demonstrate how the CBC mode hides features better than the ECB mode with DES as the base algorithm.



```
from Crypto.Cipher import DES
from Crypto import Random
from PIL import Image

def get_iv():return Random.new().read(DES.block_size)

def encrypt_image(path, path_out, cipher):
  with open(path, "rb") as f:
          image = f.read()
  header, image = image[:64], image[64:]
  image, ign = image[:(len(image) // 8) * 8], image[(len(image) // 8) * 8:]
  encrypted_image = cipher.encrypt(image)
  encrypted_image = encrypted_image + ign
  encrypted_image = header + encrypted_image
  with open(path_out, "wb") as f:
          f.write(encrypted_image)

if __name__ == '__main__':
  path = input()
  if path.endswith('.png') or path.endswith('.jpg'):
          image = Image.open(path)
          imag = image.convert('L').point(lambda x: 255 if x > 128 else 0, mode='1')
          imag.save(path[:-4] + '.bmp')
  key = bytes(input()[:8], 'ascii')
  cipher_ecb = DES.new(key, mode=DES.MODE_ECB)
  cipher_cbc = DES.new(key, DES.MODE_CBC, get_iv())
  encrypt_image(path[:-4] + '.bmp', path[:-4] + '_ecb' + '.bmp', cipher_ecb)
  encrypt_image(path[:-4] + '.bmp', path[:-4] + '_cbc' + '.bmp', cipher_cbc)
```

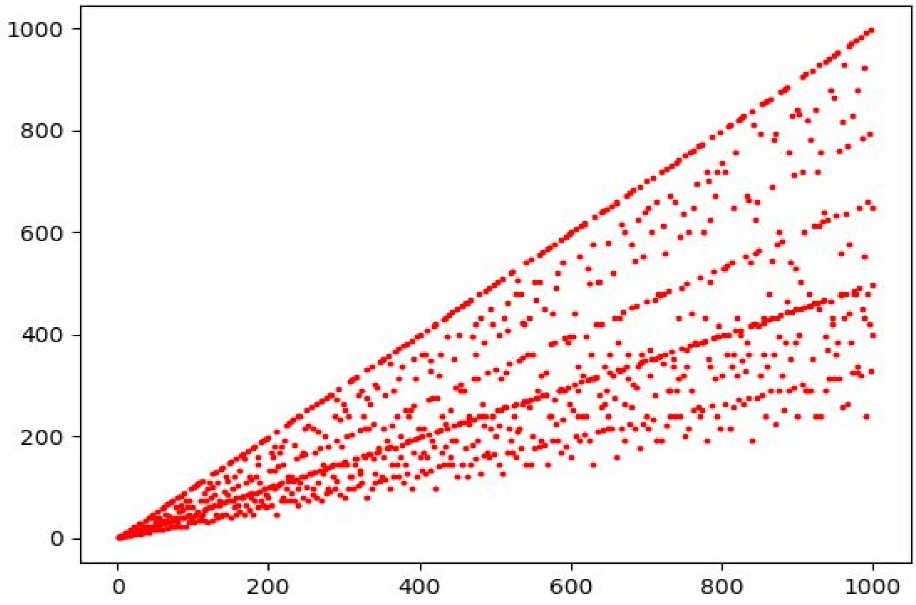
## **Practical Assignment 4:** Implement LCG, ANSI X9.17 and BBS pseudo-random number algorithms and perform 3 randomness tests mentioned in the class. Spectral test is only for LCG.



```
from math import sqrt
from time import time
from itertools import islice
from statistics import mean,stdev
from scipy.stats import chi2,norm
import numpy as np
import matplotlib.pyplot as plt
from Crypto.Cipher import DES3
from Crypto import Random
from Crypto.Util.strxor import strxor

def lcg(initial = 0, constants =[34,8], m=500 ):
    rand = initial
    a,c =constants
    while True:
            rand = (a * rand + c) % m
            yield rand / m

def ansi(initial = Random.new().read(8) , constants = [Random.new().read(16)]):
    V = initial
    key = constants[0]
    des3 = DES3.new(key, DES3.MODE_ECB)
    while True:
            EDT = des3.encrypt(hex(int(time() * 10**6))[-8:])
            R = des3.encrypt(strxor(V, EDT))
            V = des3.encrypt(strxor(R, EDT))
            yield int(V.hex(), 16)
```

```
def bbs(initial =101, constants =  [71,503]):
    s = initial
    p, q = constants
    n = p * q
    x = (s * s) % n
    while True:
            x = (x * x) % n
            b = x % 2
            yield x / n

def spectral(numbers):
    plt.scatter(numbers[1:], numbers[:-1])
    plt.show()

def count(numbers, n, r):
    ctr = 0
    for x in numbers:
            if x >= n and x <= r: ctr += 1
    return ctr


def chisquare(numbers, alpha=0.05, k=10):
    counts = []
    for i in range(k):
            counts.append(count(numbers, (i / k), (i + 1) /k))
    difference, n = 0, len(numbers)
    expected = n / k
    for i in range(k):
            err = (counts[i] - expected)**2
            difference += err / expected
    return abs(difference) >= chi2.ppf(1 - alpha, k - 1)


def ks_test(numbers):
    average = mean(numbers)
    deviation = stdev(numbers)
    n = len(numbers)
    for i in range(n):
            numbers[i] = (numbers[i] - average) / deviation
    numbers.sort()
    normal = []
    difference = []
    for i in range(n):
            normal.append(norm.cdf(numbers[i]))
            difference.append(abs((i + 1) / n - normal[i]))
    max_difference = max(difference)
    critical = 1.36 / sqrt(n)
    return max_difference >= critical


if __name__ == "__main__":
    n =1000
    prngs ,tests = ('lcg', 'bbs', 'ansi'), ('spectral', 'chisquare', 'ks_test')
    for prng in prngs:
            numbers = list(islice(vars()[prng](),n))
            for test in tests:
                    print(test,'on',prng,vars()[test](numbers))
```

## **Practical Assignment 5 :**Write a program to plot Euler's Totient Function for the first 1000 positive integers.



```
import matplotlib.pyplot as plt

def phi(n):
    result, p = n, 2
    while(p * p <= n):
                if (n % p == 0):
                                while (n % p == 0): n = int(n / p)
                                result -= int(result / p)
                p += 1
    if (n > 1): result -= int(result / n)
    return result

if __name__ == "__main__":
    plt.figure("Euler Totient Function")
    n = int(input())
    plt.scatter(range(1, n + 1), list(map(phi, range(1, n + 1))), s=3, c='r')
    plt.show()
```