Day 4

1.Write a High level code for ECB, CBC, and CFB modes, the plaintext must be a

sequence of one or more complete data blocks (or, for CFB mode, data segments). In

other words, for these three modes, the total number of bits in the plaintext must be a

positive multiple of the block (or segment) size. One common method of padding, if

needed, consists of a 1 bit followed by as few zero bits, possibly none, as are necessary to

complete the final block. It is considered good practice for the sender to pad every

message, including messages in which the final message block is already complete. What

is the motivation for including a padding block when padding is not needed?

Program:

import AES

import os

def pad(data, block\_size):

padding\_length = block\_size - (len(data) % block\_size)

padding = b'\x80' + b'\x00' \* (padding\_length - 1)

return data + padding

def unpad(data):

while data[-1] == 0:

data = data[:-1]

if data[-1] == 128:

return data[:-1]

else:

raise ValueError("Invalid padding")

def ecb\_encrypt(key, plaintext):

cipher = AES.new(key, AES.MODE\_ECB)

return cipher.encrypt(plaintext)

def ecb\_decrypt(key, ciphertext):

cipher = AES.new(key, AES.MODE\_ECB)

return cipher.decrypt(ciphertext)

def cbc\_encrypt(key, iv, plaintext):

cipher = AES.new(key, AES.MODE\_ECB)

ciphertext = b""

previous\_block = iv

for i in range(0, len(plaintext), len(key)):

block = plaintext[i:i+len(key)]

xor\_block = bytes(x ^ y for x, y in zip(block, previous\_block))

encrypted\_block = cipher.encrypt(xor\_block)

ciphertext += encrypted\_block

previous\_block = encrypted\_block

return ciphertext

def cbc\_decrypt(key, iv, ciphertext):

cipher = AES.new(key, AES.MODE\_ECB)

plaintext = b""

previous\_block = iv

for i in range(0, len(ciphertext), len(key)):

block = ciphertext[i:i+len(key)]

decrypted\_block = cipher.decrypt(block)

plaintext += bytes(x ^ y for x, y in zip(decrypted\_block, previous\_block))

previous\_block = block

return plaintext

def cfb\_encrypt(key, iv, plaintext):

cipher = AES.new(key, AES.MODE\_ECB)

ciphertext = b""

previous\_block = iv

for i in range(0, len(plaintext), len(key)):

block = plaintext[i:i+len(key)]

encrypted\_block = cipher.encrypt(previous\_block)

xor\_block = bytes(x ^ y for x, y in zip(encrypted\_block, block))

ciphertext += xor\_block

previous\_block = xor\_block

return ciphertext

def cfb\_decrypt(key, iv, ciphertext):

cipher = AES.new(key, AES.MODE\_ECB)

plaintext = b""

previous\_block = iv

for i in range(0, len(ciphertext), len(key)):

block = ciphertext[i:i+len(key)]

encrypted\_block = cipher.encrypt(previous\_block)

xor\_block = bytes(x ^ y for x, y in zip(encrypted\_block, block))

plaintext += xor\_block

previous\_block = block

return plaintext

# Example usage:

key = os.urandom(16)

iv = os.urandom(16)

plaintext = b"This is a sample plaintext."

plaintext = pad(plaintext, 16) # Add padding

print("Original plaintext:", plaintext)

# ECB mode

ecb\_ciphertext = ecb\_encrypt(key, plaintext)

ecb\_decrypted = ecb\_decrypt(key, ecb\_ciphertext)

print("ECB Decrypted:", unpad(ecb\_decrypted).decode())

# CBC mode

cbc\_ciphertext = cbc\_encrypt(key, iv, plaintext)

cbc\_decrypted = cbc\_decrypt(key, iv, cbc\_ciphertext)

print("CBC Decrypted:", unpad(cbc\_decrypted).decode())

# CFB mode

cfb\_ciphertext = cfb\_encrypt(key, iv, plaintext)

cfb\_decrypted = cfb\_decrypt(key, iv, cfb\_ciphertext)

print("CFB Decrypted:", unpad(cfb\_decrypted).decode())

2.Write a High level code for Encrypt and decrypt in cipher block chaining mode using

one of the following ciphers: affine modulo 256, Hill modulo 256, S-DES, DES. Test

data for S-DES using a binary initialization vector of 1010 1010. A binary plaintext of

0000 0001 0010 0011 encrypted with a binary key of 01111 11101 should give a binary

plaintext of 1111 0100 0000 1011. Decryption should work correspondingly

Program:

import des # You'd need a DES library

# Define your plaintext, key, and initialization vector

plaintext = "0000000100100011"

key = "0111111101"

iv = "10101010"

# Pad the plaintext to match block size

plaintext = plaintext + "0" \* (64 - len(plaintext))

# Initialize the ciphertext

ciphertext = ""

# XOR the first plaintext block with the IV

block = ""

for i in range(64):

block += str(int(plaintext[i]) ^ int(iv[i]))

ciphertext += des.encrypt(block, key) # You'd need to use your DES library here

# Loop through the rest of the blocks

for i in range(64, len(plaintext), 64):

block = ""

for j in range(64):

block += str(int(plaintext[i + j]) ^ int(ciphertext[i + j - 64]))

ciphertext += des.encrypt(block, key) # You'd need to use your DES library here

# Your ciphertext is now ready

print("Ciphertext:", ciphertext)

# Decryption works in reverse - decrypt each block and XOR with the previous ciphertext block

decrypted\_text = ""

previous\_ciphertext\_block = iv

for i in range(64, len(ciphertext), 64):

block = des.decrypt(ciphertext[i:i+64], key) # You'd need to use your DES library here

decrypted\_block = ""

for j in range(64):

decrypted\_block += str(int(block[j]) ^ int(previous\_ciphertext\_block[j]))

decrypted\_text += decrypted\_bloc

previous\_ciphertext\_block = ciphertext[i:i+64]

# Your decrypted plaintext is now ready

print("Decrypted Plaintext:", decrypted\_text)

3.Write a High level code for RSA system, the public key of a given user is e = 31,

n = 3599. What is the private key of this user? Hint: First use trial-and-error to determine

p and q; then use the extended Euclidean algorithm to find the multiplicative inverse of

31 modulo f(n).

Program:

def gcd(a, b):

if b == 0:

return a

return gcd(b, a % b)

# Given public key

e = 31

n = 3599

# Step 1: Calculate φ(n)

p = 59 # Trial-and-error to find prime factors of n

q = 61

phi\_n = (p - 1) \* (q - 1)

# Step 2: Use the Extended Euclidean Algorithm to find d

def extended\_gcd(a, b):

if a == 0:

return (b, 0, 1)

else:

gcd, x, y = extended\_gcd(b % a, a)

return (gcd, y - (b // a) \* x, x)

gcd, x, y = extended\_gcd(e, phi\_n)

d = x % phi\_n

# Ensure that d is positive

if d < 0:

d += phi\_n

# Private key is (d, n)

private\_key = (d, n)

# Print the private key

print("Private Key (d, n):", private\_key)

4.Write a High level code for set of blocks encoded with the RSA algorithm and we

don’t have the private key. Assume n = pq, e is the public key. Suppose also someone

tells us they know one of the plaintext blocks has a common factor with n. Does this help

us in any way?

Program:

import math

def gcd(a, b):

while b:

a, b = b, a % b

return a

# Given public key

e = 65537 # Typically used value for e

n = 104729 \* 104723 # n = p \* q, where p and q are prime numbers

# Encoded blocks

encoded\_blocks = [block1, block2, block3, ...] # Your encoded RSA blocks

# Someone tells us one of the plaintext blocks has a common factor with n

common\_factor\_block = encoded\_blocks[0] # Replace with the actual block

# Find the common factor (common factor block, n)

common\_factor = gcd(common\_factor\_block, n)

# Check if common factor is non-trivial

if common\_factor > 1 and common\_factor < n:

# Common factor is a non-trivial factor of n

p = common\_factor

q = n // p

phi\_n = (p - 1) \* (q - 1)

# Calculate d (private key)

d = pow(e, -1, phi\_n) # Compute modular multiplicative inverse of e modulo phi\_n

# Private key is (d, n)

private\_key = (d, n)

print("Private Key (d, n):", private\_key)

else:

print("The common factor does not help factorize n.")

5.Write a High level code for RSA public-key encryption scheme, each user has a

public key, e, and a private key, d. Suppose Bob leaks his private key. Rather than

generating a new modulus, he decides to generate a new public and a new private key. Is

this safe?

Program:

import random

# Bob's old modulus (n) and old private key (d)

old\_n = 104729 \* 104723 # Replace with Bob's old modulus

old\_d = 12345 # Replace with Bob's old private key

# Generate a new public key (e) and new private key (d) using the same modulus (n)

# Ensure that the new private key is kept secret

e = 65537 # Typically used value for e

phi\_n = (p - 1) \* (q - 1) # Calculate Euler's totient function of n

# Generate a new private key (d) as the modular multiplicative inverse of e modulo phi\_n

d = pow(e, -1, phi\_n)

# Bob's new public and private key pairs

new\_public\_key = (e, old\_n)

new\_private\_key = (d, old\_n)

print("New Public Key (e, n):", new\_public\_key)

print("New Private Key (d, n):", new\_private\_key)

6.Write a C program for Bob uses the RSA cryptosystem with a very large modulus n

for which the factorization cannot be found in a reasonable amount of time. Suppose

Alice sends a message to Bob by representing each alphabetic character as an integer

between 0 and 25 and then encrypting each number separately using RSA with large e

and large n. Is this method secure? If not, describe the most efficient attack against this

encryption method.

Program:

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

// Function to encrypt a character using RSA

unsigned long long encrypt(char plaintext, unsigned long long e, unsigned long long n) {

return fmod(pow(plaintext, e), n);

}

// Function to decrypt a character using RSA

char decrypt(unsigned long long ciphertext, unsigned long long d, unsigned long long n) {

return fmod(pow(ciphertext, d), n);

}

int main() {

// Alice sends a message to Bob

char plaintext = 'A'; // Represent 'A' as an integer between 0 and 25

unsigned long long e = 65537; // A large value of e

unsigned long long n = 104729 \* 104723; // A large modulus (n = p \* q)

// Encrypt the character

unsigned long long ciphertext = encrypt(plaintext, e, n);

// Send the ciphertext to Bob

printf("Ciphertext: %llu\n", ciphertext);

// Bob decrypts the character

char decrypted\_char = decrypt(ciphertext, d, n); // Assuming Bob has the private key

printf("Decrypted Character: %c\n", decrypted\_char);

return 0;

}