Disha Meghnani
Student ID: 2763978

Network Security and Cryptography, Summative Assessment 1
Submission Deadline 14 November 2024, Thursday 9.59 AM UK time

1. Consider the AES-128 key schedule algorithm, with the key 4247316644F4823070F4744FA232172C. Find the subkeys k_1 and k_2. **[4 points]**

AES-128 uses a key schedule to expand a single 128-bit (16-byte) initial key into 11 round keys, each also 128 bits.
These keys are used in each round of encryption, including an initial "AddRoundKey" step before the rounds begin.

Code:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

def xor_bytes(a, b):
    return bytes(x ^ y for x, y in zip(a, b))

def rot_word(word):
    return word[1:] + word[:1]

def sub_word(word):
    s_box = [
        # 0    1   2    3    4    5    6    7    8    9    A    B    C    D    E    F
        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
    ]
    return bytes([s_box[b] for b in word])

def key_expansion(key):
    RCON = [
        0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1B, 0x36, 0x6C, 0xD8, 0xAB, 0x4D, 0x9A
    ]

    expanded_keys = [key]

    for i in range(1, 11):
```

```
        temp = expanded_keys[-1][-4:]
        temp = sub_word(rot_word(temp))
        temp = xor_bytes(temp, bytes([RCON[i-1], 0, 0, 0]))

        new_key = xor_bytes(expanded_keys[-1][:4], temp)
        for j in range(1, 4):
            new_key += xor_bytes(new_key[-4:], expanded_keys[-1][j * 4:(j + 1) * 4])

        expanded_keys.append(new_key)

    return expanded_keys[1], expanded_keys[2]

# Update initial key to specified principal key in hexadecimal
initial_key = bytes.fromhex("4247316644F4823070F4744FA232172C")

k1, k2 = key_expansion(initial_key)
print(f"k1: {k1.hex()}")
print(f"k2: {k2.hex()}")
```

Output:

k1: 60b7405c2443c26c54b7b623f685a10f
k2: f585361ed1c6f4728571425173f4e35e

Functions Used:

1. **xor_bytes(a, b):** XORs two-byte arrays, which is a core operation in generating new round keys.
2. **rot_word(word)**: Rotates a 4-byte word left by one byte.
3. **sub_word(word)**: Applies the AES S-box substitution to each byte in a word. The S-box is a non-linear substitution table used for byte substitution, enhancing security.

Key Expansion (key_expansion function):

1. **Initialization**: The key_expansion function begins by storing the initial key as the first-round key (expanded_keys[0]).
2. **Round Key Generation Loop**:
    - For each of the 10 rounds, it generates a new 16-byte (128-bit) key.
    - **Transformation Step**:
        - It takes the last 4 bytes of the previous key as temp.
        - **rot_word(temp)**: Rotates temp by one byte.
        - **sub_word(temp)**: Substitutes each byte of temp using the AES S-box.
        - **XOR with RCON**: The first byte of temp is XORed with a value from RCON, a predefined table of constants, ensuring uniqueness in each round.
    - **New Key Calculation**:
        - The first 4 bytes of the new key are generated by XORing the transformed temp with the first 4 bytes of the previous key.
        - The remaining bytes are generated by XORing each successive 4-byte block with the corresponding block of the previous key.

Result:

- After completing the loop, the function returns k1 and k2, the first two round keys after the initial key.

2. The file at https://canvas.bham.ac.uk/files/17789736/download is encrypted using AES-128 in CTR mode, using:

   Key = 2B75B85C6CD84D0AB432D228ADC2060D
   nonce (a.k.a. IV) = 85C2B686921E512AFF0DB3C67F6D8D97.

   Decrypt it and describe the image you get.
   Hint. The plaintext file is an image in JPG format. To decrypt, use any suitable software or online tool you like. One option is openssl. **[4 points]**

Decryption using the openssl command-line tool:

1. Download the encrypted file to your local machine.
2. Prepare the decryption command with OpenSSL, specifying AES-128 in CTR mode with the provided key and nonce.

Command for decryption:
openssl enc -aes-128-ctr -d -in Ex2_2_ciphertext -out decrypt.jpg -K 2B75B85C6CD84D0AB432D228ADC2060D -iv 85C2B686921E512AFF0DB3C67F6D8D97

- **-aes-128-ctr:** specifies the AES algorithm in CTR mode.
- **-d:** indicates decryption.
- **-in:** is the input file (your encrypted file).
- **-out:** is the output file, which will be the decrypted image in JPG format.
- **-K:** is the encryption key.
- **-iv:** is the nonce (initialization vector).

3. After running it, you should get a file named decrypt.jpg.
4. Open the decrypted file to view the image.

Command to open the image:
feh decrypt.jpg

Output:


(50 KB file)
(Output is an image of penguin with red muffler, yellow beak and feet.)

3. In 60 words or less, explain why an encryption system that satisfies IND-CPA can't be broken by machine learning. **[4 points]**

An encryption system with IND-CPA guarantees that any two chosen plaintexts produce indistinguishable ciphertexts, even under computational scrutiny. This unpredictability prevents machine learning models from discerning patterns or correlations between inputs and outputs, making reverse-engineering or accurate plaintext prediction infeasible regardless of data volume or model sophistication.

4. Find a printable string *x* such that the first 20 bits of SHA-256(*x*) are zero.
   Note: the x you submit must be different from the x that everyone else submits. **[4 points]**

   To find a unique string x such that the first 20 bits of SHA-256(x) are zero, we can use a brute-force method to generate candidate strings and check their SHA-256 hashes until we find a match.

   Code:

```
import hashlib
import random
import string

def find_unique_string():
    while True:
        # Generate a random printable string x
        x = ''.join(random.choices(string.printable, k=10))

        # Compute SHA-256 hash
        sha256_hash = hashlib.sha256(x.encode()).hexdigest()

        # Check if the first 20 bits are zero
        if sha256_hash.startswith("00000"):
            return x, sha256_hash

x, sha256_hash = find_unique_string()
print(f"String x: {x}")
print(f"SHA-256(x): {sha256_hash}")
```

Output:

String x: "ix*8wl,f.
SHA-256(x): 00000c7d497aca779d51ebfac9b12349475783134d18fffe8c5193c48dc96529

This code randomly generates a printable string x, computes its SHA-256 hash, and checks if the first 20 bits (5 hex characters) are zero. It repeats this process until a matching x is found. This approach ensures that each execution yields a unique result by relying on randomness for x.

5. Fred proposes to define a new hash function which takes an input of up to 2^{32} bits, as follows:

```
INPUT: bitstring message m of length < 2**32 bits
Pad m with a 32-bit encoding of the length of m
c := empty string
    while there are still bits to take from m:
        Take the next bit from m and append it to c
        c := c << 1
  Take the next |c| bits from m and xor them into c
Truncate c to 64 bits, if necessary
OUTPUT: c
```

Here, |c| means the length of c, and << 1 means rotate the string left by one bit. If there are insufficiently many bits to take from m, then we pad m with 0s. Prove to Fred that his idea is inadequate, by finding a collision for Fred's proposed function. **[4 points]**

Fred's Hash Function:

It works by taking an input bitstring m (up to 2^{32} bits in length) and performing bitwise operations to produce a 64-bit output hash c.

The function is vulnerable because:

1. **Small outputs:** A 64-bit output is too small for a secure hash function for messages as large as 2^{32} bits, making it prone to collisions.
2. **Bitwise manipulations:** The way c is updated (rotation and XOR with message bits) does not sufficiently mix the bits of m, meaning similar inputs may produce similar outputs.

Code:

```
import random

def freds_hash_function(m):
    # Convert the message `m` (binary string) to the appropriate padded format
    bitstring = m

    # Step 1: Pad with 32-bit encoding of the length of m
    length = len(bitstring)
    length_bitstring = format(length, '032b')
    padded_message = bitstring + length_bitstring

    # Step 2: Initialize `c` as an empty bitstring
    c = ''

    # Step 3: Process each bit in the message according to Fred's function
    idx = 0
    while idx < len(padded_message):
        # Rotate `c` left by 1 bit if not empty
        if c:
            c = c[1:] + c[0]

        # Append the next bit from padded_message to `c`
        if idx < len(padded_message):
            c += padded_message[idx]
            idx += 1
```

```
        # XOR the next `|c|` bits from `padded_message` into `c`
        length_c = len(c)
        for i in range(length_c):
            if idx + i < len(padded_message):
                # XOR operation with padding if necessary
                c = c[:i] + str(int(c[i]) ^ int(padded_message[idx + i])) + c[i + 1:]

        # Move the index forward by |c| bits
        idx += length_c

        # Truncate `c` to 64 bits if necessary
        if len(c) > 64:
            c = c[:64]

    # Return the final `c` as a binary string
    return c if c else '0' * 64

# Function to find a collision
def find_collision():
    hash_table = {}

    while True:
        # Generate a random 32-bit binary message
        m = ''.join(random.choice('01') for _ in range(32))  # Random 32-bit binary string

        # Compute the hash
        hash_value = freds_hash_function(m)

        # Check if this hash has already been seen with a different message
        if hash_value in hash_table:
            original_message = hash_table[hash_value]
            if original_message != m:  # Confirm it's a distinct message
                print("Collision found!")
                print("Message 1 (binary):", original_message)
                print("Message 2 (binary):", m)
                print("Hash (binary):", hash_value)
                return original_message, m, hash_value
        else:
            # Store the hash and the message if it's new
            hash_table[hash_value] = m

# Run the function to find a collision
find_collision()
```

Output:

Collision found!
Message 1 (binary): 11010010111011101011011000110000
Message 2 (binary): 00000110010011001100011001111101
Hash (binary): 1100101000
('11010010111011101011011000110000',
'00000110010011001100011001111101',
'1100101000')

Fred's Hash Function:

- Pads the message with the 32-bit length encoding.
- Performs the left rotation, appends the next bit, and then applies XOR with bits from the message.
- Truncates c to 64 bits as specified.

Collision Detection:

- Uses a dictionary to store each unique hash.
- Generates random messages and checks for distinct messages with the same hash.

6. The “ciphers.zip” file (linked from the Canvas page) contains some text files with RSA PKCS encrypted ciphertext. The files can be decrypted using the secret key key.pem (linked from the Canvas page) and openssl rsautl.

(a) What is the modulus *N* of the secret key in key.pem. **[2 point]**

(b) Decrypt the file matching the last 4 digits of your student id and write down the result. **[3 point]**

a) Modulus N:

The modulus N in RSA is part of the public and private keys and is used for encryption and decryption.

We can obtain N from key.pem using the OpenSSL command-line tool.

Command:

openssl rsa -in key.pem -noout -modulus

Output:

Modulus=B86DF078C3301C0FDBE142E2D721039DD6CAE4428E5D50EF2504A5
549F8CA6A100BFE9FD56877FE8CFF4886556935C1149D3E699C3085EB499097
9A064997E7CCF284195633B82D25C7F9BDD1F128FE72E8A5C8B75C63E3935A
433E5592A1EB735EB04678AA9FE44AE1987D4C88156C5C0CB7A2C8C86782D
A3EE2E082A63E20F2D5F88A1C2E9E10D67DC0B120E4C6D5814149C108EFA3
84F5D1DE4ABAD59DED63C21AFCB3F5B03AF043515FC5C135405712511D07E
B37547F603CA7F62063330CE772F4DC07E5D8DE196449C668EA57FA9092488D
2FA72B9F6FA8455D71D4381116C5C8F18F9F7EFA9C922067BA8D4A6A1C03A
740E880FEEEA06849CC56B959B

b) Decrypting cipher file:

Find the file corresponding to the last 4 digits of your student ID.

- My student ID is 2763978 so last 4 digits are 3978, hence I took cipher3978.txt.

Use OpenSSL to decrypt the file with the pkeyutl command:

- Uses the private key in key.pem to decrypt the file cipher3978.txt.
- Outputs the decrypted content to decrypted.txt.

Command:

openssl pkeyutl -decrypt -inkey key.pem -in ./ciphers/cipher3978.txt -out decrypted.txt

Read the Decrypted Output:

- Open decrypted.txt to see the decrypted message

Command:

cat decrypted.txt

Output:

89dfa55474d2fe15

7. Consider the following RSA based encryption scheme that we will call BadModPKC. H_1 and H_2 are two hash functions mapping onto $\mathbb{Z}_n^*$.

Procedure $\mathsf{Keygen}(1^\lambda)$

01 : Choose two random $\lambda/2$-bit primes p and q
02 : $n = p \cdot q$
03 : $\phi = (p-1)(q-1)$
04 : Select e such that $1 < e < \phi$ and $\gcd(e, \phi) = 1$
05 : Compute d such that $1 < d < \phi$ and $ed \equiv 1 \pmod{\phi}$
06 : Set $PK = (e, n)$
07 : Set $SK = (d)$
08 : **return** (PK, SK)

Procedure $\mathsf{Encrypt}(PK, m)$

// We assume $m \in \mathbb{Z}_n^*$
01 : $r \xleftarrow{\$} \mathbb{Z}_n^*$
02 : $c_1 = r^e \bmod n$
03 : $c_2 = m \cdot H_1(r) \bmod n$
04 : $c_3 = r \cdot H_2(m) \bmod n$
05 : **return** $c = (c_1, c_2, c_3)$

Show that BadModPKC is not IND-CPA secure. **[5 point]**

The IND-CPA security notion (indistinguishability under chosen-plaintext attack) means that given the encryption of either of two chosen messages, an adversary should not be able to tell which message was encrypted.
However, this "BadModPKC" scheme is vulnerable because of the way it uses the randomness r in the encryption process.
In a secure scheme, if two encryptions are different, the ciphertexts should ideally look random and reveal no correlation with the plaintexts.
Here, c3 is derived directly from r and the hash H2(m), which potentially leaks information about the plaintext.

Attack Outline:

1. **Adversary Setup**: The adversary picks two messages m0 and m1.
2. **Challenge Query**: The adversary submits m0 and m1 to the challenger, who randomly encrypts one of them using the "BadModPKC" scheme and sends back the ciphertext.
3. **Distinguishing Strategy**: The adversary uses the values of c1, c2, and c3 to guess which message was encrypted.

Steps of Attack:

- Let m0=1 and m1≠1 be two messages chosen by the adversary.
- Since m0=1, this simplifies c2 to c2=H1(r) when m0 is encrypted.
- When m0=1:
  - c2 = m0·H1(r) = H1(r)mod n.
  - c3 = r·H2(m0) = r·H2(1)mod n.
- Compute r:
  $r = \frac{c3}{H2(m)} \bmod n$
- Compute c2:
  $c2 = H1\left(\frac{c3}{H2(1)}\right) \bmod n$
- Now there is no dependency on r
- Hence the adversary can compute c2 and verify if it matches with c2 of m0 if it is same then message m0 is encrypted else message m1

Pseudo Code:

m_0 ← 1
m_1 ← some value in Z_n* such that m_1 ≠ 1
b ← {0, 1} // Challenger randomly picks b = 0 or 1
c = (c_1, c_2, c_3) ← Encrypt(PK, m_b) // Challenger returns ciphertext
if c2 = H1($\frac{c3}{H2(1)}$) mod n
  guess b = 0 // Guess that m_0 = 1 was encrypted
else:
  guess b = 1 // Guess that m_1 was encrypted