Part 1

1 Block and Stream Ciphers [4.5 points]

1. We consider the implementation of a cipher with a 128-bit key. An attacker manages to recover the value of k key bits using side-channel information. Does this info help her to attack the cipher? If yes, what would be the attacker's gain over the complexity of an exhaustive search?

2. An attacker mounts a partial key recovery attack on a cipher with a 128-bit key and manages to find a system of k independent linear equations involving the secret key bits with $k < n$. For example:

$$k_1 \oplus k_2 \oplus k_{63} = 1$$
$$k_1 \oplus k_3 \oplus k_{17} = 1$$
$$k_{43} \oplus k_{56} \oplus k_{60} = 0$$

   Does this info help her to attack the cipher? If yes, what would be the attacker's gain over the complexity of an exhaustive search?

3. Consider a block cipher in counter mode as a way to construct a stream cipher. What is its state, and what is its state-update function?

1. Yes. Since the attacker has recovered *k* bits, then he needs to exhaustively search the remaining (128-k) bits. Thus, the attacker's gain is $2^{(128-k)}$ over the complexity of an exhaustive search, which is $O(2^{128})$. The more *k* bits the attacker has recovered, the bigger the reduction in the search space would be.
2. Yes. The more *k* independent linear equations the attacker has found, the more *k* bits he can recover, and thus, the bigger the part of the secret key he can recover. The attacker's gain is 2^k over the complexity of an exhaustive search, which is $O(2^{128})$.
3. State: input block as a counter and nonce (initialization vector) — if nonce is random, it's added to the counter with concatenation, XOR, or addition; if nonce is not random, it's concatenated to the counter.
   State-update function: The current state (generally given as plaintext) is encrypted using the block cipher. This generates the next keystream block. The output is a ciphertext block.

Part 2

2 Hash Functions [4.5 points]

1. Consider a 64-bit hash function $H(x)$. Describe an efficient algorithm allowing you to find a pair of messages M, M' such that $H(M) = H(M')$. Describe what resources (i.e. time and memory complexity) it would require to find such a pair.

2. Given an n-bit cryptographic hash function H, how many messages should we expect to hash before finding a message x such that $H(x) < 2^{n-k}$ for some integer k?

3. Alice uses the secret key k to generate a MAC of message m by computing $MAC(k, m) = H(m) \oplus k$, where H is a secure hash function. Is this a secure or insecure MAC? Briefly explain your answer.

1. The Birthday paradox/attack finds a pair of two different messages *M* and *M'* that produce the same hash value *H(M)=H(M')* by finding a collision in a hash function after hashing $2^{64/2}=2^{32}$ different messages. Time complexity is $O(2^{64/2})=O(2^{32})$. Memory space is also $O(2^{64/2})=O(2^{32})$.
2. This is a collision problem in which we search for a hash value *H(x)* that falls below the threshold of 2^{n-k}. The probability of collision in such hash table is $1-[(n^2-n)/2(2^{n-k})]$, i.e., we need to hash $1-[(n^2-n)/2(2^{n-k})]$ different messages before finding a message *x*.
3. This is an insecure MAC. There is no verification of the MAC. Also, the secret key *k* is just one, which means it's reused for all messages. Furthermore, the secret key is directly XORed, so if an attacker recovers the MAC or the hash function, he can recover the secret key by XORing what he's already obtained.

Part 3

3 Assessing the Security of a Cipher [4 points]

Consider a cipher which, for each bit of the plaintext p, uses a random coin toss to flip the bit (in which case the corresponding ciphertext bit c is given by $c = p + 1 \bmod 2$) or to leave it unchanged (i.e. $c = p$). Thus, half of the bits are left unencrypted. Is this cipher secure? Briefly explain your answer.

The cipher is not secured. Bit flipping in the plaintext leads to corruption of all the following ciphertext blocks, as the decryption will be incorrect. Further, we have that bits are flipped by a random coin toss, and that half of the bits are left unencrypted. Thus, an attacker can simply reverse the encryption by bruteforcing both possibilities, 0 or 1, for each bit. There's no encryption key nor verification of this cipher.

Part 4

4 AES Round Transformations [7 points]

The file `simple_aes.c` (available on Moodle) contains a simple AES128 implementation written in the C programming language. However, the four AES round transformations are not contained in the source code and still need to be implemented. Then, these four round transformations have to be applied to the plaintext block `0123456789abcdeffedcba9876543210` as shown in the function `aes_round_trans`, which corresponds to encryption with a round-reduced version of AES128 consisting of just a single round. The round key (in hex format) is `0f1571c947d9e8590cb7add6af7f6798`, but the last four bytes `af7f6798` need to be replaced by the last four bytes of your student-ID number (see comments at the beginning of the function `aes_round_trans` for an example).

In order to get the full points for this question, the correct ciphertext (after applying the four round transformations as shown in `aes_round_trans`) as well as the source code of the implementation of the four round transformations have to be submitted. Please copy the source code of the four round transformations into the PDF file that contains the answers of this homework, i.e. do not attach them as a separate file. There is no strict requirement to write the implementation of the round transformations in C, i.e. using another language like Java or Python is permitted. However, it is not allowed to use a library that provides the AES round transformations.

My Student ID: 0232420949 = 0xdda7655 in hex

```
round_key = [0x0f, 0x15, 0x71, 0xc9, 0x47, 0xd9, 0xe8, 0x59, 0x0c, 0xb7, 0xad, 0xd6, 0x0d, 0xda, 0x76, 0x55]
# Addition of round key
# byte_matrix[4][4]
# key_matrix[4][4]
def aes_add_round_key(byte_matrix, key_matrix):
    for r in range(0, 4):
        for c in range(0, 4):
            byte_matrix[r][c] ^= key_matrix[r][c] #Performs bitwise XOR
```

```
# SubBytes transformation
# byte_matrix[4][4]
def aes_sub_bytes(byte_matrix):
    for r in range(0, 4):
        for c in range(0, 4):
            byte_matrix[r][c] = sbox_table[byte_matrix[r][c]]
```

```
# ShiftRows transformation
# byte_matrix[4][4]
def aes_shift_rows(byte_matrix):
    for r in range(1, 4): #No rotation on first row
        #row 2 rotates 1 byte: leftmost goes to rightmost
        #row 3 rotates 2 bytes and row 4 rotates 3 bytes
        byte_matrix[r] = byte_matrix[r][r:] + byte_matrix[r][:r]
```

```
# MixColumns transformation
# byte_matrix[4][4]
def aes_mix_columns(byte_matrix):
    galois_filed_matrix = [
        [0x02, 0x03, 0x01, 0x01],
        [0x01, 0x02, 0x03, 0x01],
        [0x01, 0x01, 0x02, 0x03],
        [0x03, 0x01, 0x01, 0x02]
    ]
    for c in range(0, 4):
        temp = [0] * 4 #Temp column
        for r in range(0,4):
            for i in range(0,4):
                temp[r] ^= gf256_mul(byte_matrix[i][c], galois_filed_matrix[r][i]) #Performs bitwise
XOR
        for r in range(0,4):
            byte_matrix[r][c] = temp[r]
```

```
Output:
plaintext :
0x0123456789abcdeffedcba9876543210

roundkey  :
0x0f1571c947d9e8590cb7add60dda7655

state-matrix:
01 89 fe 76
23 ab dc 54
```

```
45 cd ba 32
67 ef 98 10
```

```
key-matrix  :
0f 47 0c 0d
15 d9 b7 da
71 e8 ad 76
c9 59 d6 55
```

```
after add_round_key:
0e ce f2 7b
36 72 6b 8e
34 25 17 44
ae b6 4e 45
```

```
after sub_bytes:
ab 8b 89 21
05 40 7f 19
18 3f f0 1b
e4 4e 2f 6e
```

```
after shift_rows:
ab 8b 89 21
40 7f 19 05
f0 1b 18 3f
6e e4 4e 2f
```

```
after mix_columns:
13 73 74 5d
4e bc dd 45
a2 f5 72 2b
8a 31 1d 07

ciphertext:
0x134ea28a73bcf53174dd721d5d452b07
```

Works Consulted

Biryukov, Alex. "Introduction to Symmetric Cryptography." Information Security Basics, 2023, University of Luxembourg, Luxembourg. PowerPoint presentation.

Menezes, Alfred J., et al. *Handbook of Applied Cryptography*. CRC Press, 2001.

For Question 2.2 I used the formula for average probability of total number of collisions from: Kiao, Ue. "Probability of Collision in Hash Function [Complete Analysis]." *OpenGenus IQ: Computing Expertise & Legacy*, OpenGenus IQ: Computing Expertise & Legacy, 20 Mar. 2022, iq.opengenus.org/probability-of-collision-in-hash/.

For Question 4 I used: Daemen, Joan & Rijmen, Vincent. "A Specification for Rijndael, the AES Algorithm", https://moodle.uni.lu/pluginfile.php/912735/mod_resource/content/2/aes.spec.v316.pdf