# IDATT2503 - Cryptography Assignment 03

## Problem 1 - Hash and MAC

> Describe the role of cryptographic hash functions and MACs. How do they differ?

Cryptographic hash functions is an algorithm that takes in a data input, which may vary in length/size, and then based on the input will produce an output of a non-variable length, i.e. a given hashing algorthm will always produce an output of that given length regardless of the length of input. This makes it impossible to deduce the length/size of the input from the length/size of the output. A good hashing function must have both collision resistance, where no two different inputs can produce the same output, and preimage resistance, which means that one should in practice be unable to reverse-engineer the output to find the input.

The role of a hash function is often as a cryptographic signature, enabling us to verify data integrity, and to store data like passwords securely. Being deterministic, a given input will always produce the same hash output. Thus, the hash of a file will always be the same, which is how we can use hashing to verify that a file has not been changed from when created and hashed by a trusted authority, publisher or contact.

MAC (Message Authentication Code) is a tool used to verify theintegrity of messages. It uses cryptographich hashing together with a secret key in order to create a cryptographically unique signature. Where hashing offers "only" data integrity,  MAC, by using a unique key, offers authentication of specific users in addition to verifying integrity of data.

## Problem 2 - LFSR

Solving for LFSR:

- Start with the keys and simulate the sequence bit by bit, until we arrive back at the starting condidion in order to find the period.
- Initialize LFSR with the given key and use the state update equation to generate the next bit
- Continue until the sequence repeats.

### 2.a

First Linear Feedback Shift Register (LFSR) defined by:

$$z_{i+4} = z_i + z_{i+1} + z_{i+2} + z_{i+3} (mod 2)$$

Keys:

$$1 : K = 1000$$
$$2 : K = 0011$$

Periods:

```
Period for key 1000 with LFSR a): 5, sequence: [1, 0, 0, 0, 1]

Period for key 0011 with LFSR a): 5, sequence: [0, 0, 1, 1, 0]
```

### 2.b

Second LFSR defined by:

$$z_{i+4} = z_i + z_{i+3}(mod2)$$

Keys:

$$1 : K = 1000$$
$$2 : K = 0011$$

Periods:

```
Period for key 1000 with LFSR b): 15, sequence: [1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0,
0]

Period for key 0011 with LFSR b): 15, sequence: [0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1,
0]
```

## Problem 3 - $GF(2^8) field$

### 3.a

```
# GF(2^8) multiplication modulo polynomial x^8 + x^4 + x^3 + x + 1 (00011011)
MODULO_POLY = 0x1B

def gf_multiply(a, b):
    result = 0
    for _ in range(8):  # Up to 8 bits in a
        if b & 1:  # If the lowest bit of b is 1
            result ^= a  # XOR with a
        carry = a & 0x80  # Check if there is a carry bit
        a <<= 1  # Shift a left
        if carry:  # Perform reduction if needed
            a ^= MODULO_POLY
        b >>= 1  # Shift b right
    return result & 0xFF  # Mask result to 8 bits

# Load the multiplication table from the file
import numpy as np

with open("mul_table.txt") as file:
    mul_table = np.loadtxt(file, dtype=int)
```

```
# Check and print all results against the multiplication table
for a in range(256):
    for b in range(256):
        calculated_result = gf_multiply(a, b)
        table_result = mul_table[a][b]
        print(f"a={a}, b={b}: calculated={calculated_result}, table={table_result}")

        if calculated_result != table_result:
            print(f"Mismatch found at a={a}, b={b}: calculated={calculated_result} !=
table={table_result}")

print("Verification completed.")
```

Output:

```
50: calculated=201, table=201
a=150, b=51: calculated=95, table=95
a=150, b=52: calculated=144, table=144
a=150, b=53: calculated=6, table=6
a=150, b=54: calculated=167, table=167
a=150, b=55: calculated=49, table=49
a=150, b=56: calculated=34, table=34
a=150, b=57: calculated=180, table=180
a=150, b=58: calculated=21, table=21
a=150, b=59: calculated=131, table=131
a=150, b=60: calculated=76, table=76
[snipp]
a=255, b=246: calculated=85, table=85
a=255, b=247: calculated=170, table=170
a=255, b=248: calculated=216, table=216
a=255, b=249: calculated=39, table=39
a=255, b=250: calculated=61, table=61
a=255, b=251: calculated=194, table=194
a=255, b=252: calculated=9, table=9
a=255, b=253: calculated=246, table=246
a=255, b=254: calculated=236, table=236
a=255, b=255: calculated=19, table=19
Verification completed.
```

## 3.b

Code:

```
# Load the multiplication table from the file
import numpy as np

with open("mul_table.txt") as file:
    mul_table = np.loadtxt(file, dtype=int)

# Find multiplicative inverses for all elements different from zero
inverses = {}

for a in range(1, 256):  # Start from 1 since 0 has no multiplicative inverse
    for b in range(1, 256):
        if mul_table[a][b] == 1:
            inverses[a] = b
            break

# Print the results
for a in range(1, 256):
    if a in inverses:
        print(f"Multiplicative inverse of {a} is {inverses[a]}")
    else:
        print(f"No inverse found for {a}")

print("All inverses found.")
```

For every non-zero value $a$ from $1$ to $255$, the code iterated through $b$ to find a $b$ such that $\mathrm{mul\_table[a][b]} = 1$. This means that $b$ is the multiplicative inverse of $a$.

Output:

```
Multiplicative inverse of 1 is 1
Multiplicative inverse of 2 is 141
Multiplicative inverse of 3 is 246
Multiplicative inverse of 4 is 203
Multiplicative inverse of 5 is 82
Multiplicative inverse of 6 is 123
[snipp]
Multiplicative inverse of 250 is 125
Multiplicative inverse of 251 is 160
Multiplicative inverse of 252 is 205
Multiplicative inverse of 253 is 26
Multiplicative inverse of 254 is 65
Multiplicative inverse of 255 is 28
All inverses found.
```

## 3.c

Using the following code to determine that the multiplication $GF(2^8)$ is commutative, associative and distributive:

```
# GF(2^8) multiplication modulo polynomial x^8 + x^4 + x^3 + x + 1 (00011011)
MODULO_POLY = 0x1B

def gf_multiply(a, b):
    result = 0
```

```python
    for _ in range(8):  # Up to 8 bits in a
        if b & 1:  # If the lowest bit of b is 1
            result ^= a  # XOR with a
        carry = a & 0x80  # Check if there is a carry bit
        a <<= 1  # Shift a left
        if carry:  # Perform reduction if needed
            a ^= MODULO_POLY
        b >>= 1  # Shift b right
    return result & 0xFF  # Mask result to 8 bits

# Check commutative property: ab = ba
commutative = True
for a in range(256):
    for b in range(256):
        if gf_multiply(a, b) != gf_multiply(b, a):
            print(f"Commutative property failed for a={a}, b={b}")
            commutative = False

# Check associative property: (ab)c = a(bc)
associative = True
for a in range(256):
    for b in range(256):
        for c in range(256):
            if gf_multiply(gf_multiply(a, b), c) != gf_multiply(a, gf_multiply(b, c)):
                print(f"Associative property failed for a={a}, b={b}, c={c}")
                associative = False

# Check distributive property: a(b + c) = ab + ac
distributive = True
for a in range(256):
    for b in range(256):
        for c in range(256):
            left = gf_multiply(a, b ^ c)
            right = gf_multiply(a, b) ^ gf_multiply(a, c)
            if left != right:
                print(f"Distributive property failed for a={a}, b={b}, c={c}")
                distributive = False

# Print the overall result
if commutative:
    print("Commutative property holds for all a, b in GF(2^8)")
else:
    print("Commutative property does not hold for all a, b in GF(2^8)")

if associative:
    print("Associative property holds for all a, b, c in GF(2^8)")
else:
    print("Associative property does not hold for all a, b, c in GF(2^8)")

if distributive:
    print("Distributive property holds for all a, b, c in GF(2^8)")
else:
    print("Distributive property does not hold for all a, b, c in GF(2^8)")
```

Output:

```
Commutative property holds for all a, b in GF(2^8)
Associative property holds for all a, b, c in GF(2^8)
Distributive property holds for all a, b, c in GF(2^8)
```

## 3.d

To find some element $g$ in $GF(2^n)$ such that the sequence $g, g^2, g^3, ..., g^{255}$ consists of all the non-zero elements in GF $(2^n)$, $g$ must be a *generator* for the multipliocative group $GF(2^8)$. Such an element would have the order $255$, which means that $g^{255} = 1$, and that all other powers of $g$ will yeild unique non-zero elements prior to this point.

Solution:

- Iterate through all elements of $GF(2^8)$.
- Calculate the sequence $g^1, g^2, ..., g^{255}$ for every element $g$.
- Verify that the sequence consists of all unique non-zero element in the field.

Code to find such an element, i.e. generator, if present:

```
# GF(2^8) multiplication modulo polynomial x^8 + x^4 + x^3 + x + 1 (00011011)
MODULO_POLY = 0x1B

def gf_multiply(a, b):
    result = 0
    for _ in range(8):  # Up to 8 bits in a
        if b & 1:  # If the lowest bit of b is 1
            result ^= a  # XOR with a
        carry = a & 0x80  # Check if there is a carry bit
        a <<= 1  # Shift a left
        if carry:  # Perform reduction if needed
            a ^= MODULO_POLY
        b >>= 1  # Shift b right
    return result & 0xFF  # Mask result to 8 bits

# Function to calculate g^k in GF(2^8)
def gf_pow(g, k):
    result = 1
    for _ in range(k):
        result = gf_multiply(result, g)
    return result

# Find a generator g
def find_generator():
    for g in range(1, 256):  # Skip 0, as it has no inverse or generator property
        seen = set()
        for power in range(1, 256):
            seen.add(gf_pow(g, power))
        if len(seen) == 255:  # All non-zero elements found
            return g
    return None
```

```
generator = find_generator()
if generator:
    print(f"Element {generator} is a generator for GF(2^8)")
else:
    print("No generator found in GF(2^8)")
```

Output:

```
Element 3 is a generator for GF(2^8)
```

## Problem 4 - CTR-mode

Code

```
# Problem 4a

# GF(2^8) multiplication modulo polynomial x^8 + x^4 + x^3 + x + 1 (00011011)
MODULO_POLY = 0x1B

def gf_multiply(a, b):
    result = 0
    for _ in range(8):  # Up to 8 bits in a
        if b & 1:  # If the lowest bit of b is 1
            result ^= a  # XOR with a
        carry = a & 0x80  # Check if there is a carry bit
        a <<= 1  # Shift a left
        if carry:  # Perform reduction if needed
            a ^= MODULO_POLY
        b >>= 1  # Shift b right
    return result & 0xFF  # Mask result to 8 bits

# Find the multiplicative inverse in GF(2^8)
def gf_inverse(x):
    for i in range(1, 256):
        if gf_multiply(x, i) == 1:
            return i
    return None  # Should not happen if x != 0

# Key and nonce values
key = int('01001010', 2)
nonce = int('0110', 2) << 4  # Shift 4 bits to represent the 4-bit nonce

# Generate the first 4 bytes of the key-stream
key_stream = []
for counter in range(4):  # First 4 values of the counter (0 to 3)
    x = nonce | counter  # Concatenate the nonce and the counter
    x_inv = gf_inverse(x)  # Find the multiplicative inverse
    if x_inv is not None:
        key_stream_byte = x_inv ^ key  # XOR with the key
        key_stream.append(key_stream_byte)
```

```python
# Print the first 4 bytes of the key-stream
print("First 4 bytes of the key-stream:")
for i, byte in enumerate(key_stream):
    print(f"Byte {i + 1}: {byte:08b} (decimal: {byte})")
```

## 4.a

```python
# Key and nonce values
key = int('01001010', 2)
nonce = int('0110', 2) << 4  # Shift 4 bits to represent the 4-bit nonce
```

Output:

```
First 4 bytes of the key-stream:
Byte 1: 01011100 (decimal: 92)
Byte 2: 00010100 (decimal: 20)
Byte 3: 11100101 (decimal: 229)
Byte 4: 10011001 (decimal: 153)
```

## 4.b

In CTR,, theperiod to a key-stream will depend on how many different values the counter and nonce can generate. Since we have a 4-bit nonce and a 4-bit counter, the maximal period will be

$$2^4 \cdot 2^4 = 256$$

different combinations, i.e. one for each byte vlaue of $GF(2^8)$.

## 4.c

Yes, the computation of the keystream can easily be paralellized. This is one of the advantages of CRT for encryption. Since every block is being encrypted using a unique value (combination of nonce and counter), these values can be computed independently since they're not dependent upon the result of the block prior nor after it.

## Problem 5 - HMAC

HMAC definition:

$$HMAC(K, m) = h((K \oplus opad) \parallel h((K \oplus ipad) \parallel m))$$

where

- $K$ is the key
- $ipad$ and $opad$ are padding values.
- $h$ is the midsquare hashing functino.

## 5.a

Find the HMAC for the message $0110$.

Input values:

- $K = 1001$
- $ipad = 0011$
- $opad = 0101$
- Message $m = 0110$

Steps to solve

Step 1: Beregn $K \oplus ipad$:

$$1001 \oplus 0011 = 1010$$

Step 2: Concatenate message $m$

$$(K \oplus ipad) \parallel m = 1010 \parallel 0110 = 10100110$$

Step 3: Hash $h((K \oplus ipad||m))$

$$10100110^2(mod2^8) \to 1 + 1 + +11 +_2 = 166_{10}$$
$$166^2 \implies 27556(mod256) = 100$$
$$100_{10} = 01100100_2$$

$100_{10} = 01100100_2$, where the middle four bits are $1001$.

Thus, $h((K \oplus ipad||m)) = 1001$

Step 4: Compute $K \oplus opad$

$$1001 \oplus 0101 = 1100$$

Step 5: Concatenate result from step 4 with result from step 3

$$(K \oplus opad)||h((K \oplus ipad)||m) = 1100||1001 = 11001001$$

Step 6: Hash $h((K \oplus opad)||h((K \oplus ipad)||m))$

$$11001001^2(mod2^8) = 11001001_2 = 201_{10}$$
$$201^2 = 40401 \implies 4041(mod256) = 145$$
$$145_{10} = 10010001_2$$

$145_{10} = 10010001_2$, where the middle four bits are $0100$.

Thus, $h((K \oplus ipad||h((K \oplus ipad)m)) = 1001$

Finally we have that the HMCA for the message $m(0110) = 0100$

Code:

```python
# Problem 5

def midsquare_hash(x):
    # Beregn x^2 modulo 2^8
    squared = (x ** 2) % 256
    # Konverter til binær representasjon med 8 bits lengde
    binary_representation = f"{squared:08b}"
    # Trekk ut de midterste fire bitene
    middle_four = binary_representation[2:6]
    # Returner de midterste fire bitene som en heltallsverdi
    return int(middle_four, 2)

# Definer inngangsverdier
key = int('1001', 2)
ipad = int('0011', 2)
opad = int('0101', 2)
message = int('0110', 2)

# Steg 1: Beregn K xor ipad
k_ipad = key ^ ipad

# Steg 2: Concat K xor ipad med meldingen og beregn hash
combined_ipad_message = (k_ipad << 4) | message  # Shifter K xor ipad og legger til
meldingen
hash1 = midsquare_hash(combined_ipad_message)

# Steg 3: Beregn K xor opad
k_opad = key ^ opad

# Steg 4: Concat K xor opad med resultatet fra forrige hash og beregn ny hash
combined_opad_hash1 = (k_opad << 4) | hash1
hmac_result = midsquare_hash(combined_opad_hash1)

# Skriv ut resultatet
print(f"HMAC for the message '0110' is: {hmac_result:04b} (decimal: {hmac_result})")
```

Output:

```
HMAC for the message '0110' is: 0100 (decimal: 4)
```

## 5.b

> You receive the message 0111, with HMAC 0100. Is it reason to believe that the message is authenic?

Yes, because we are using a very weak hashing function, which means that the chance of collisions are quite a lot higher than if using a proper hashing function than in the assignment.

## Problem 6

### 6.a

The purpose of the Initialization Vector (IV) in CBC-mode (Cipher Block Chaining) is to ensure that identical message content does not lead to identically encrypted block every time the same message is encrypted using the same key. This adds random variation to the encryption process, so that even when two identical messages are encrypted using the same key, the encrypted output will not be identical, given that they're using different IVs.

### 6.b

If using IV as a salt instead of as a fixed IV when generating CBC-MAC, an attacker can exploit this to fake valid CBC-MAC based on known messages and their MACs. This means that if an attacker have access to a valid pair $x, H_{IV}(x)$, where $x$ is the messange and $H_{IV}(x)$ is its corresponding CBC-MAC, and the IV is known in plaintext, they can use this to construct new valid MAC pairs.

To construct a new message, $x'$:

$$x' = IV1 \oplus IV2 \oplus x$$

This manipulation works because during the computation of $x'$ with IV2, the `XOR` operation involving the original IV1 and new IV2 cancels out, effectively neutralizing the change. This leaves the rest of the message $x$ to be processed as if it were the original message, generating a valid MAC for $x'$ with IV2.

This demonstrates why a fixed IV is crucial in CBC-MAC, as using a variable IV as a salt allows attackers to create forged messages and their valid CBC-MACs.