Modular Arithmetic, AES Encryption, and Hash Analysis Lab

Parameters:

A: SimonasRiska

B: 20185536

C: VILNIUSGEDIMINASTECHNIKALUNIVERSITY

Task I:

- **Calculate greatest common divisor for B1 and B2 (divide B into two parts B=B1||B2)**

I wrote this Python code solution for this task:

gcd-for-b1-b2.py

C: > Users > simon > Desktop > Cryptography > gcd-for-b1-b2.py > ...

```python
def gcd_extended(a, b):
    if a == 0:
        return b, 0, 1

    gcd, x1, y1 = gcd_extended(b % a, a)

    x = y1 - (b // a) * x1
    y = x1

    return gcd, x, y

B = 20185536

B1 = int(str(B)[:4])
B2 = int(str(B)[4:])

gcd_b1_b2, _, _ = gcd_extended(B1, B2)

print(f"GCD of B1 and B2 (where B1 = {B1} and B2 = {B2}): {gcd_b1_b2}")
```

Results:

```
C:\Users\simon\Desktop\Cryptography>python gcd-for-b1-b2.py
GCD of B1 and B2 (where B1 = 2018 and B2 = 5536): 2
```

Here, my student number (B) was split into two equal parts (first four digits of B to form B1, last four digits of B to form B2). As B = 20185536, B1 = 2018 and B2 = 5536.

Then I wrote *gcd_extended* function to calculate the greatest common divisor of those two integers – B1 and B2. This function also calculates coefficients x and y for equation $a \times x + b \times y =$

$GCD(a, b)$. Here, the algorithm is used to compute the GCD of B1 = 2018 and B2 = 5536. The algorithm operates recursively by replacing the first number by second until the first number becomes zero. It starts with *gcd_extended(2018, 5536)*, then calls *gcd_extended(5536 % 2018, 2018)* which is equal to *gcd_extended(1500, 2018)* and continues this process until reaching the case *gcd_extended(0, 2)* where GCD is 2. When recursion unwinds the function calculates the coefficients x and y by back-substituting the results from each recursive step, providing values that satisfy that $2018 \times x + 5536 \times y = 2$.

Answer: 2.

- **Calculate inverse B mod 661**

I edited the Python code used for finding greatest common divisor to calculate inverse 20185536 mod 661:

inverse_b_mod_661.py

C: > Users > simon > Desktop > Cryptography > inverse_b_mod_661.py > ...

```python
def gcd_extended(a, b):
    if a == 0:
        return b, 0, 1
    gcd, x1, y1 = gcd_extended(b % a, a)
    x = y1 - (b // a) * x1
    y = x1
    return gcd, x, y

B = 20185536

modulus = 661
gcd, inverse_b, _ = gcd_extended(B, modulus)

if gcd == 1:
    inverse_b = inverse_b % modulus
    print(f"Modular inverse of B modulo {modulus}: {inverse_b}")
else:
    print(f"No modular inverse exists for B modulo {modulus} since GCD(B, {modulus}) ≠ 1")
```

Results:

```
C:\Users\simon\Desktop\Cryptography>python inverse_b_mod_661.py
Modular inverse of B modulo 661: 266
```

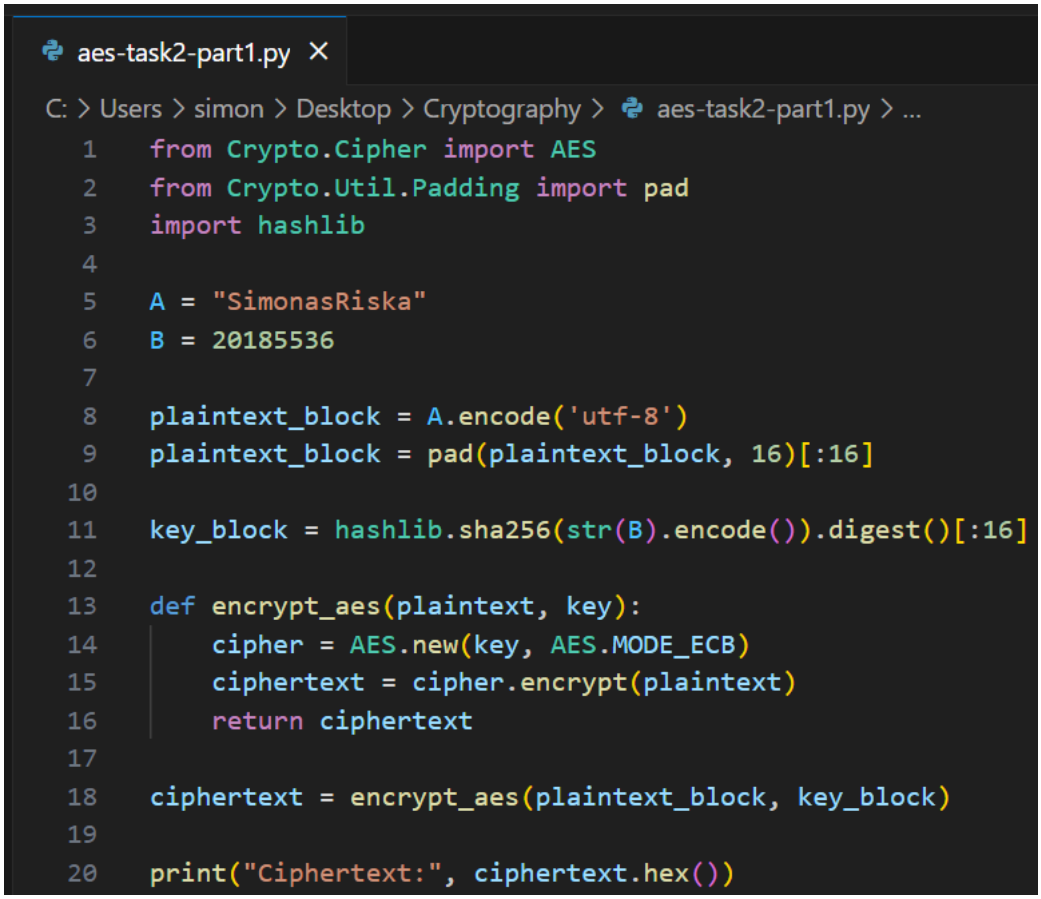
Here, the function uses the previously defined function gcd_extended which implements the Extended Euclidean Algorithm, the GCD with integers 20185536 mod 661. The GCD of 20185536 and 661, as well as the coefficients that meet the formula $20185536 \times x + 661 \times y = GCD(20185536, 661)$ are found by calling this function. 20185536 and 661 are coprime if the GCD is 1, which will ensure that there is a modular inverse. 20185536 The inverse of mod 661 is represented by the algorithm's coefficient inverse_b, which can be negative. The code computes the inverse_b % modulus to ensure that the inverse is a positive integer in the allowed range [0; 660] . Finally, the program prints the modular inverse. If the GCD is not 1 then the code correctly identifies that no modular inverse exists and prints it.

Answer: 266.

Task II:

- **Create 128-bits block from A (padding or truncating) to use as a plaintext block, create 128-bits block from B (padding or truncating) to use as a key and encrypt using AES**

I wrote and used this Python code:



I got this result:

```
C:\Users\simon\Desktop\Cryptography>py aes-task2-part1.py
Ciphertext: af173ff698a7a07826019f78f88208d6
```

Here, the string “SimonasRiska” is encoded to UTF-8 bytes. Then I applied padding using the pad function from *Crypto.Util.Padding* truncating it to 16 bytes. For the encryption key I converted 20185536 to string, then encoded it to bytes, then byte representation is hashed using SHA-256 from which I extracted the first 16 bytes to form a 128-bit key suitable for AES. Then I created *encrypt_aes* function to initialize AES cipher in ECB mode. The ciphertext is displayed in hexadecimal format for readability.

Answer: af173ff698a7a07826019f78f88208d6

- **Switch one bit in plaintext block and encrypt with the same key, switch on bit in the key and encrypt the same block,**
  Calculate how many bits changed their value in the ciphertext in both cases

I wrote and used this Python code:

aes-task2-part2.py

C: > Users > simon > Desktop > Cryptography > aes-task2-part2.py > ...

```python
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
import hashlib

A = "SimonasRiska"
B = 20185536

plaintext_block = A.encode('utf-8')
plaintext_block = pad(plaintext_block, 16)[:16]

key_block = hashlib.sha256(str(B).encode()).digest()[:16]

def encrypt_aes(plaintext, key):
    cipher = AES.new(key, AES.MODE_ECB)
    ciphertext = cipher.encrypt(plaintext)
    return ciphertext

original_ciphertext = encrypt_aes(plaintext_block, key_block)

def count_bit_differences(bytes1, bytes2):
    diff_count = 0
    for b1, b2 in zip(bytes1, bytes2):
        diff_count += bin(b1 ^ b2).count('1')
    return diff_count

modified_plaintext_block = bytearray(plaintext_block)
modified_plaintext_block[0] ^= 0b00000001

ciphertext_modified_plaintext = encrypt_aes(modified_plaintext_block, key_block)

modified_key_block = bytearray(key_block)
modified_key_block[0] ^= 0b00000001

ciphertext_modified_key = encrypt_aes(plaintext_block, modified_key_block)

bit_diff_plaintext = count_bit_differences(original_ciphertext, ciphertext_modified_plaintext)

bit_diff_key = count_bit_differences(original_ciphertext, ciphertext_modified_key)

print("Original ciphertext:", original_ciphertext.hex())
print("Ciphertext with Modified Plaintext:", ciphertext_modified_plaintext.hex())
print("Ciphertext with Modified Key:", ciphertext_modified_key.hex())
print(f"Bit difference after modifying plaintext: {bit_diff_plaintext} bits")
print(f"Bit difference after modifying key: {bit_diff_key} bits")
```

I got these results:

```
C:\Users\simon\Desktop\Cryptography>py aes-task2-part2.py
Original ciphertext: af173ff698a7a07826019f78f88208d6
Ciphertext with Modified Plaintext: fe5c23edf531cd45acfdc995d13d958f
Ciphertext with Modified Key: fd79f5f90fbd1d677fbe8d0b53720abb
Bit difference after modifying plaintext: 71 bits
Bit difference after modifying key: 68 bits
```

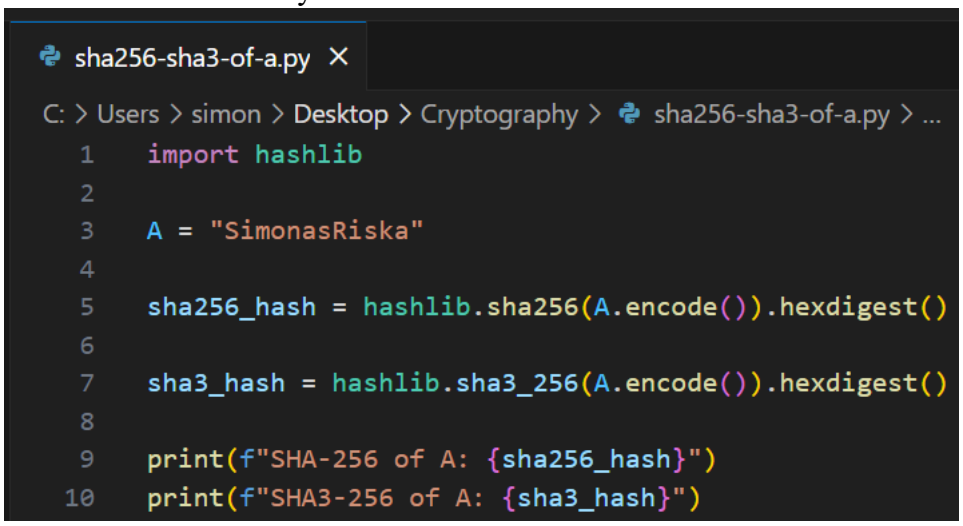
After getting original ciphertext I modified a single bit in the plaintext block, saved it into variable *modified_plaintext_block* and re-encrypted it with *encrypt_aes* function using the same key which was saved in variable *key_block*. Then I altered a single bit in encryption key, saved it into variable *modified_key_block* and encrypted original plaintext again with *encrypt_aes* function. To compare the new ciphertext with the original I wrote a function *count_bit_differences,* it iterates through each pair of bytes from the original and modified ciphertexts by performing a XOR operation. It shows how many bits have changed their value, I used it for ciphertexts and keys.

Answer: After modifying plaintext – 71 bits, after modifying key – 68 bits.

Task III:

- **Calculate SHA256 and SHA3 of A**

I wrote and used this Python code:



I got these results:

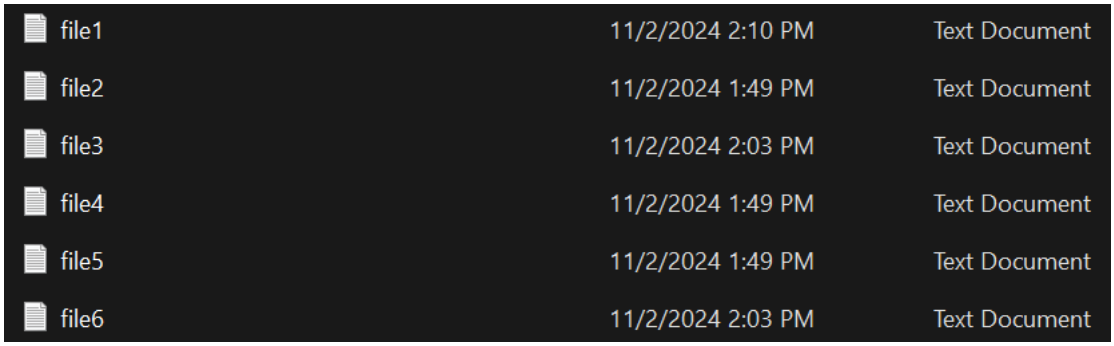
```
C:\Users\simon\Desktop\Cryptography>py sha256-sha3-of-a.py
SHA-256 of A: 49726b40213fe1b2e12c3a6f487e9872a211a91742c8b28b47603c3b0df6214b
SHA3-256 of A: 3c9f11f79da1e7cac55f4ba3f23522f4d67fb008695268 4dcd8765f67e8e1b40
```

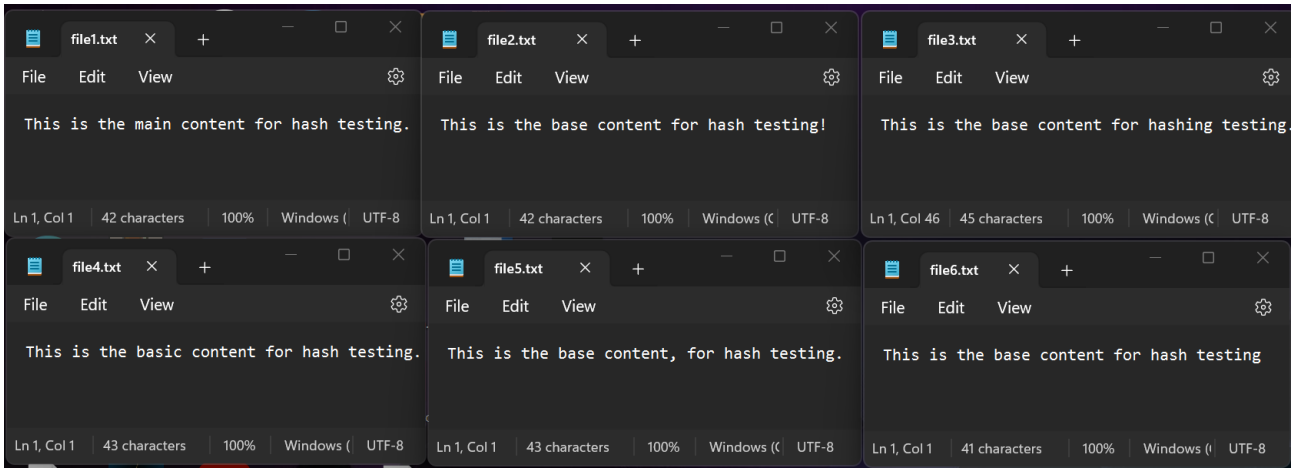
A.encode() converted the string “SimonasRiska” into bytes, then *hashlib.sha256()* computed the SHA-256 hash and *.hexdigest()* converted it to a hexadecimal string for readability. *hashlib.sha3_256()* computed the SHA3-256 hash also displayed in hexadecimal format.

Answer: SHA-256 of A: 49726b40213fe1b2e12c3a6f487e9872a211a91742c8b28b47603c3b0df6214b, SHA3-256 of A: 3c9f11f79da1e7cac55f4ba3f23522f4d67fb0086952684dcd8765f67e8e1b40

- **Find two different files with the same last octal symbol of the hash value**

I created files with different contents:





Then I wrote and used this Python code:

files-same-last-octal-symbol-hash.py

C: > Users > simon > Desktop > Cryptography > files-same-last-octal-symbol-hash.py > ...

```python
import hashlib

files = ["file1.txt", "file2.txt", "file3.txt", "file4.txt", "file5.txt", "file6.txt"]

def calculate_sha256(filename):
    with open(filename, 'rb') as f:
        content = f.read()
    sha256_hash = hashlib.sha256(content).hexdigest()
    return sha256_hash

def get_last_octal_symbol(sha256_hash):
    last_byte = int(sha256_hash[-2:], 16)
    last_octal_symbol = format(last_byte & 0b111, '03b')
    return last_octal_symbol

results = {}
for file in files:
    sha256_hash = calculate_sha256(file)
    last_octal_symbol = get_last_octal_symbol(sha256_hash)
    results[file] = last_octal_symbol
    print(f"File: {file}")
    print(f"SHA-256: {sha256_hash}")
    print(f"Last octal symbol of SHA-256: {last_octal_symbol}\n")

octal_groups = {}
for file, octal_symbol in results.items():
    octal_groups.setdefault(octal_symbol, []).append(file)

for octal_symbol, group in octal_groups.items():
    if len(group) > 1:
        print(f"Files with the same last octal symbol '{octal_symbol}': {group}")
```

I got these results:

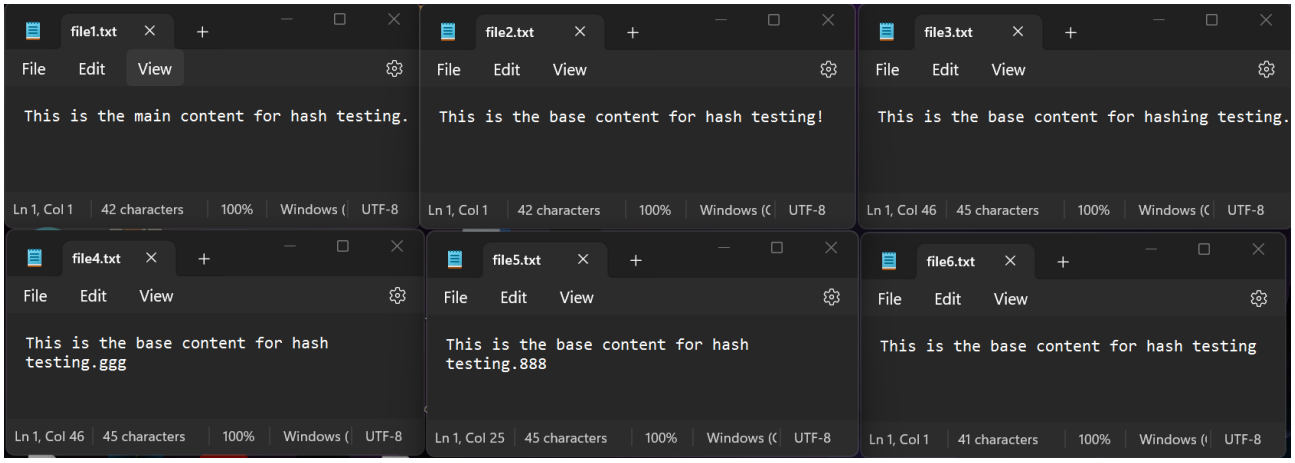
```
C:\Users\simon\Desktop\Cryptography>py files-same-last-octal-symbol-hash.py
File: file1.txt
SHA-256: 5307e0fed1d11acdec35237431dbf718541dbb11416f04c5a61fd089e925a6d3
Last octal symbol of SHA-256: 011

File: file2.txt
SHA-256: a560d80d535a58f6ff90d6fe192af165bddbfb2337a9830ba74844076169b6b0
Last octal symbol of SHA-256: 000

File: file3.txt
SHA-256: 6a4ad50eb73b88b01c2fa2c14e8d719ada0dc8ceb9a25a4ae59b1b0a5ff5b484
Last octal symbol of SHA-256: 100

File: file4.txt
SHA-256: a34ec083a34a2c4397c9f7efe772f827617fee6cc6267d68e600dcadb3cbbd02
Last octal symbol of SHA-256: 010

File: file5.txt
SHA-256: 86133f1dc096bedf578e5da6a69732d565c706facf76fbde01435a22d02b0867
Last octal symbol of SHA-256: 111

File: file6.txt
SHA-256: 24e6b1de3a2f19c685baf432e4759ba3b44d29d033790f2fe303ede9c16bf324
Last octal symbol of SHA-256: 100

Files with the same last octal symbol '100': ['file3.txt', 'file6.txt']
```

In the code I created a list names *files* containing the filenames to be analyzed. *calculate_sha256* function reads the content of each file and computes the SHA-256 hash. The *get_last_octal_symbol* function takes two characters of SHA-256 hash string which represents the last byte and converts it to integer and then it extracts the last 3 bits which represents the last octal symbol. The code groups files by their last octal symbols using a dictionary where each unique octal symbol points to a list of files. If particular octal symbol has more than one file it indicates a match and the program prints the files with the same last octal symbol.

Answer: Two different files with the same last octal symbol were found – file3.txt (content – “This is the base content for hashing testing.”) and file6.txt (content – “This is the base content for hash testing”).

- **Find two different files with the same last byte of the hash values for SHA256 and SHA3**

I used these files for this task:



I wrote and used this Python code:

two-files-same-last-byte.py

C: > Users > simon > Desktop > Cryptography > two-files-same-last-byte.py > ...

```python
import hashlib

files = ["file1.txt", "file2.txt", "file3.txt", "file4.txt", "file5.txt", "file6.txt"]

def calculate_hashes(filename):
    with open(filename, 'rb') as f:
        content = f.read()
    sha256_hash = hashlib.sha256(content).hexdigest()
    sha3_hash = hashlib.sha3_256(content).hexdigest()
    return sha256_hash, sha3_hash

def get_last_byte(sha256_hash, sha3_hash):
    last_byte_sha256 = int(sha256_hash[-2:], 16)
    last_byte_sha3 = int(sha3_hash[-2:], 16)
    return last_byte_sha256, last_byte_sha3

results = {}
for file in files:
    sha256_hash, sha3_hash = calculate_hashes(file)
    last_byte_sha256, last_byte_sha3 = get_last_byte(sha256_hash, sha3_hash)
    results[file] = (last_byte_sha256, last_byte_sha3)
    print(f"File: {file}")
    print(f"SHA-256: {sha256_hash}")
    print(f"SHA3-256: {sha3_hash}")
    print(f"Last byte of SHA-256: {last_byte_sha256}")
    print(f"Last byte of SHA3-256: {last_byte_sha3}\n")

matching_files = []
for file1, (last_byte_sha256_1, last_byte_sha3_1) in results.items():
    for file2, (last_byte_sha256_2, last_byte_sha3_2) in results.items():
        if file1 != file2 and last_byte_sha256_1 == last_byte_sha256_2 and last_byte_sha3_1 == last_byte_sha3_2:
            matching_files.append((file1, file2))
            break

if matching_files:
    for file1, file2 in matching_files:
        print(f"Files '{file1}' and '{file2}' have matching last bytes in both SHA-256 and SHA3-256 hashes.")
else:
    print("No matching files found.")
```

I got these results:

```
C:\Users\simon\Desktop\Cryptography>py two-files-same-last-byte.py
File: file1.txt
SHA-256: 5307e0fed1d11acdec35237431dbf718541dbb11416f04c5a61fd089e925a6d3
SHA3-256: f90d6dd2e2c5b14359ed3a41edd8ca3083c23cd4489c00bbe289c08d8ecb8839
Last byte of SHA-256: 211
Last byte of SHA3-256: 57

File: file2.txt
SHA-256: a560d80d535a58f6ff90d6fe192af165bddbfb2337a9830ba74844076169b6b0
SHA3-256: 7c388d50c48e7a6750cb6c1f4ab5bb087a289578df69f2964f6b3b4721803112
Last byte of SHA-256: 176
Last byte of SHA3-256: 18

File: file3.txt
SHA-256: 6a4ad50eb73b88b01c2fa2c14e8d719ada0dc8ceb9a25a4ae59b1b0a5ff5b484
SHA3-256: 6c3d6957b1e3f020e27b3950ad58ffb7d546be5b06615e027a72ef6f765fee64
Last byte of SHA-256: 132
Last byte of SHA3-256: 100

File: file4.txt
SHA-256: c6f0085b13453327eb99964f8c0b714a72d72faa7b14535b67467ae71ed06273
SHA3-256: 1daabee6600af9961663e3d3ec78da9e898ee43c85c7d6942086f1b12cc223a9
Last byte of SHA-256: 115
Last byte of SHA3-256: 169

File: file5.txt
SHA-256: 4bbf90269f7ed28526072d9f8bee80a1bff93b73be1034d6b986a760a9eac873
SHA3-256: 23649fded421f50201e3e58c6ee8d8c0fcddbf5c3ba8014953ca4f94df395fa9
Last byte of SHA-256: 115
Last byte of SHA3-256: 169

File: file6.txt
SHA-256: 24e6b1de3a2f19c685baf432e4759ba3b44d29d033790f2fe303ede9c16bf324
SHA3-256: 81c3f96b23abe1291829ea2d80a844e3630476aa02d52101471d2cfc46b524f7
Last byte of SHA-256: 36
Last byte of SHA3-256: 247

Files 'file4.txt' and 'file5.txt' have matching last bytes in both SHA-256 and SHA3-256 hashes.
Files 'file5.txt' and 'file4.txt' have matching last bytes in both SHA-256 and SHA3-256 hashes.
```

In this code I declared *files* array with filenames that I want to analyze. The *calculate_hashes* function read the file's binary content and computes its SHA-256 and SHA3-256 hashes and it returns their hexadecimal representations. *get_last_byte* function extracts the last byte from each hash and converts the final two hexadecimal characters to an integer. These last bytes are stored in the *results* dictionary and it maps each filename to a tuple of its last bytes from SHA-256 and SHA3-256 results. Then the script iterates through the *results* to identify and print pairs of files that share identical last bytes in both their SHA-256 and SHA3-256 hashes.

Answer: Files file4.txt (content – "This is the base content for hash testing.ggg") and file5.txt (content – "This is the base content for hash testing.888").