Assignment #2

20192973 김태현

Q1. Secure Hash Algorithm (SHA) is one kind of popular hash function, where SHA-256, SHA-384, and SHA-512 algorithms can produce hash values with 256, 384, and 512 bits in length, respectively. Please explain why we usually say SHA-256, SHA-384, and SHA-512 algorithms are designed to match the security of AES with 128, 192, and 256 bits, respectively.

1.Collision Resistance

For cryptographic hash functions, an essential property is collision resistance, which means it should be computationally difficult to find two distinct inputs that hash to the same output. For a hash function producing an n-bit output, a generic method to find a collision, called the birthday attack, requires about 2^(n/2) operations. This is because of the birthday paradox from probability theory, which states that in a set of roughly 2^(n/2) randomly chosen values, there's a some pair of them will be the same with high chance.

SHA-256 has 256-bit output, which means a collision resistance of 2^128. SHA-384 and SHA-512 have 384-bit and 512-bit output, providing collision resistance of 2^192 and 2^256, respectively.

2.Brute-Force Attacks

AES is a symmetric encryption algorithm, the security of AES against a brute-force attack is proportional to its key length. An attacker would need to try about 2^n different keys for an n-bit key length to break the encryption.

AES-128, AES-192 and AES-256 need 2^128, 2^192 and 2^256 operations, respectively to break via brute-force.

3. Comparison

When we compare the security strengths,

The SHA-256's 2^128 operations for collision resistance matches the brute-force strength of AES-128.

SHA-384's 2^192 operations for collision resistance matches the strength of AES-192.

Finally, SHA-512's 2^256 operations for collision resistance matches to the strength of AES-256.

Security refers to the computational effort required to overcome the security mechanism. From the above comparison, we can know why SHA-256, SHA-384, and SHA-512 are designed to match the security of AES with 128, 192, and 256 bits respectively. It's an approximation based on the operations required for collision resistance in hash functions and brute-force resistance in symmetric encryption.

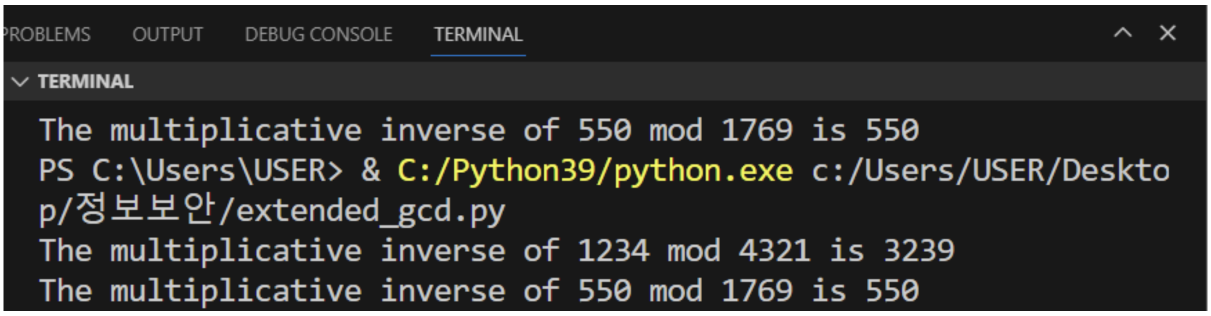
Q2. Using Euclid's gcd theorem, determine the following. Show the complete process.
(a) gcd(24140,16762)

gcd(24140, 16762) = gcd(16762, 24140%16762) = gcd(16762, 7378) = gcd(7378, 16762%7378) = gcd(7378, 2006) = gcd(2006, 7378%2006) = gcd(2006, 1360) = gcd(1360, 2006%1360) = gcd(1360, 646) = gcd(646, 1360%646) = gcd(646, 68) = gcd(68, 646%68) = gcd(68, 34) = gcd(34, 68%34) = gcd(34, 0) = 34 // stop : gcd(24140, 16762) = 34

Q3. Using the "extended" Euclidean algorithm, find the multiplicative inverse of the following. Show the complete process with all columns. If you used a programming code, include the code and the output.

Code of extended Euclidean algorithm :

```
def extended_gcd(a, m):
    x0, x1, y0, y1 = 1, 0, 0, 1
    og_a, og_m = a, m  #to print result
    while m != 0:
        q, a, m = a // m, m, a % m
        x0, x1 = x1, x0 - q * x1
        y0, y1 = y1, y0 - q * y1

    if a != 1:
        print("Inverse doesn't exist!")
    else:
        print(f"The multiplicative inverse of {og_a} mod {og_m} is {x0 %
og_m}")

# Test
extended_gcd(1234,4321)
extended_gcd(550, 1769)
```



```
The multiplicative inverse of 550 mod 1769 is 550
PS C:\Users\USER> & C:/Python39/python.exe c:/Users/USER/Deskto
p/정보보안/extended_gcd.py
The multiplicative inverse of 1234 mod 4321 is 3239
The multiplicative inverse of 550 mod 1769 is 550
```

(a) 1234 mod 4321

The multiplicative inverse of 1234 mod 4321 is 3239

(b) 550 mod 1769

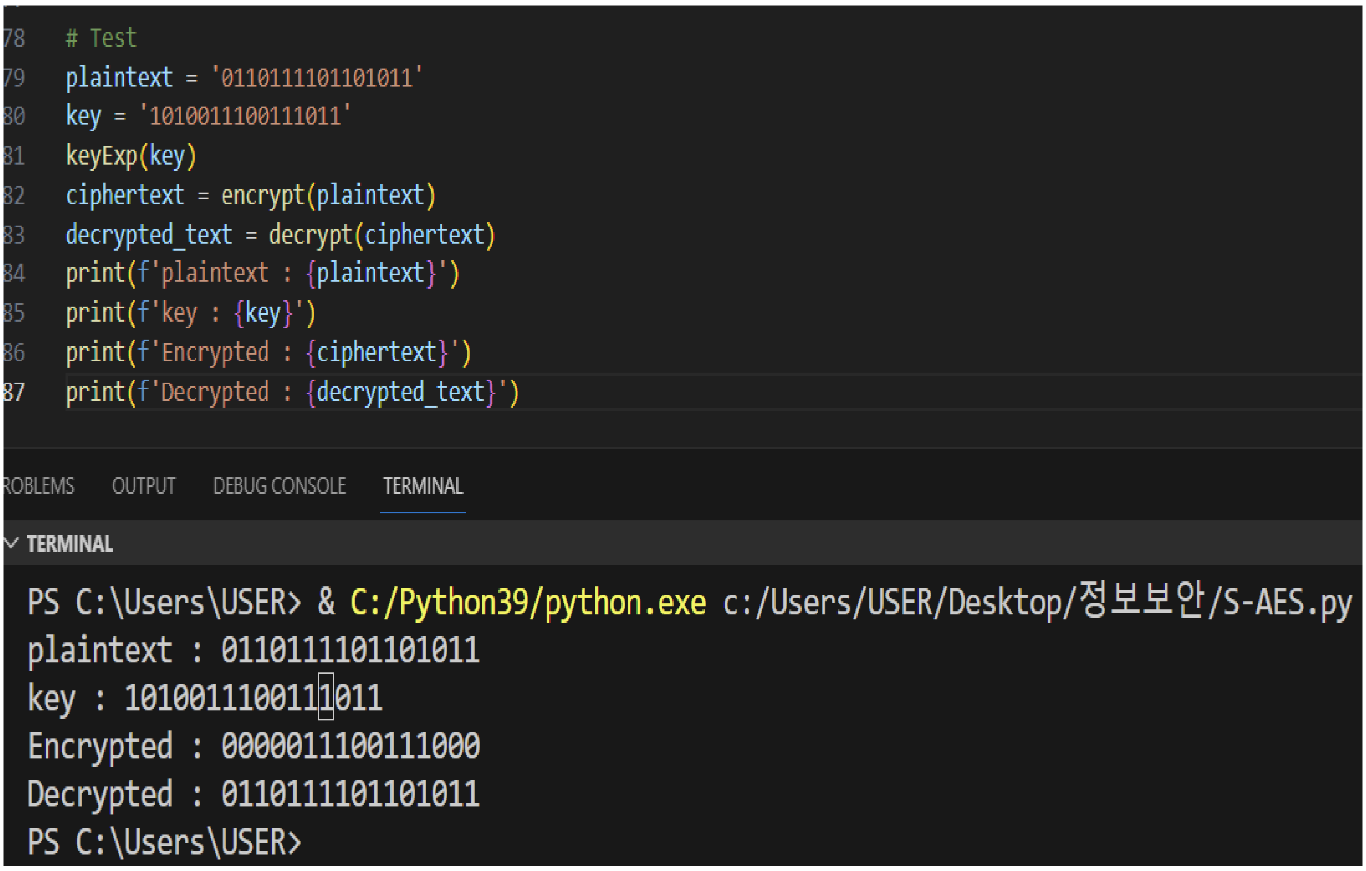
The multiplicative inverse of 550 mod 1769 is 550

Q4.

Write a programming code that can encrypt and decrypt using S-AES (Simplified AES) [1-4]. Test data: A binary plaintext of 0110 1111 0110 1011 encrypted with a binary key of 1010 0111 0011 1011 should give a binary ciphertext of 0000 0111 0011 1000. Decryption should work correspondingly. Provide the code and the execution proof. You will need to look up some additional information, such as the Galois field, for your implementation.

```python
sBox  = [0x9, 0x4, 0xa, 0xb, 0xd, 0x1, 0x8, 0x5,
         0x6, 0x2, 0x0, 0x3, 0xc, 0xe, 0xf, 0x7]

sBoxI = [0xa, 0x5, 0x9, 0xb, 0x1, 0x7, 0x8, 0xf,
         0x6, 0x0, 0x2, 0x3, 0xc, 0x4, 0xd, 0xe]

w = [None] * 6

def bin_to_int(bin_str):
    return int(bin_str, 2)

def int_to_bin(int_val, padding=16):
    return format(int_val, f'0{padding}b')

def mult(p1, p2):
    p = 0
    while p2:
        if p2 & 0b1:
            p ^= p1
        p1 <<= 1
        if p1 & 0b10000:
            p1 ^= 0b11
        p2 >>= 1
    return p & 0b1111

def intToVec(n):
    return [n >> 12, (n >> 4) & 0xf, (n >> 8) & 0xf, n & 0xf]

def vecToInt(m):
    return (m[0] << 12) + (m[2] << 8) + (m[1] << 4) + m[3]

def addKey(s1, s2):
    return [i ^ j for i, j in zip(s1, s2)]

def sub4NibList(sbox, s):
    return [sbox[e] for e in s]
```

```python
def shiftRow(s):
    return [s[0], s[1], s[3], s[2]]

def keyExp(key):
    key = bin_to_int(key)
    Rcon1, Rcon2 = 0b10000000, 0b00110000
    w[0] = (key & 0xff00) >> 8
    w[1] = key & 0x00ff
    w[2] = w[0] ^ Rcon1 ^ sub2Nib(w[1])
    w[3] = w[2] ^ w[1]
    w[4] = w[2] ^ Rcon2 ^ sub2Nib(w[3])
    w[5] = w[4] ^ w[3]

def sub2Nib(b):
    return sBox[b >> 4] + (sBox[b & 0x0f] << 4)

def encrypt(ptext):
    ptext = bin_to_int(ptext)
    def mixCol(s):
        return [s[0] ^ mult(4, s[2]), s[1] ^ mult(4, s[3]),
                s[2] ^ mult(4, s[0]), s[3] ^ mult(4, s[1])]
    state = intToVec(((w[0] << 8) + w[1]) ^ ptext)
    state = mixCol(shiftRow(sub4NibList(sBox, state)))
    state = addKey(intToVec((w[2] << 8) + w[3]), state)
    state = shiftRow(sub4NibList(sBox, state))
    return int_to_bin(vecToInt(addKey(intToVec((w[4] << 8) + w[5]), state)))

def decrypt(ctext):
    ctext = bin_to_int(ctext)
    def iMixCol(s):
        return [mult(9, s[0]) ^ mult(2, s[2]), mult(9, s[1]) ^ mult(2, s[3]),
                mult(9, s[2]) ^ mult(2, s[0]), mult(9, s[3]) ^ mult(2, s[1])]

    state = intToVec(((w[4] << 8) + w[5]) ^ ctext)
    state = sub4NibList(sBoxI, shiftRow(state))
    state = iMixCol(addKey(intToVec((w[2] << 8) + w[3]), state))
    state = sub4NibList(sBoxI, shiftRow(state))

    return int_to_bin(vecToInt(addKey(intToVec((w[0] << 8) + w[1]), state)))

# Test
plaintext = '0110111101101011'
key = '1010011100111011'
keyExp(key)
ciphertext = encrypt(plaintext)
decrypted_text = decrypt(ciphertext)
print(f'plaintext : {plaintext}')
print(f'key : {key}')
print(f'Encrypted : {ciphertext}')
print(f'Decrypted : {decrypted_text}')
```



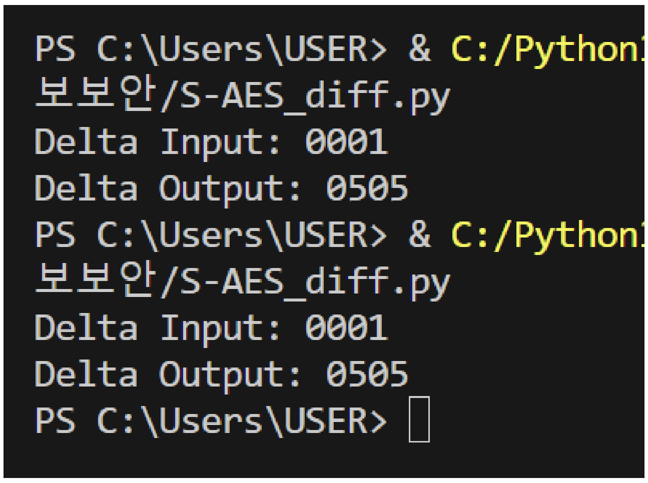
```python
# Test
plaintext = '0110111101101011'
key = '1010011100111011'
keyExp(key)
ciphertext = encrypt(plaintext)
decrypted_text = decrypt(ciphertext)
print(f'plaintext : {plaintext}')
print(f'key : {key}')
print(f'Encrypted : {ciphertext}')
print(f'Decrypted : {decrypted_text}')
```

```
PS C:\Users\USER> & C:/Python39/python.exe c:/Users/USER/Desktop/정보보안/S-AES.py
plaintext : 0110111101101011
key : 1010011100111011
Encrypted : 0000011100111000
Decrypted : 0110111101101011
PS C:\Users\USER>
```

Q5. Implement a differential cryptanalysis attack on 1-round S-AES. Explain the logic behind the attack. Show all steps.

```python
# Define the S-Box for SubNibbles
S_BOX = {
    '0': '9', '1': '4', '2': 'A', '3': 'B',
    '4': 'D', '5': '1', '6': 'E', '7': 'F',
    '8': '2', '9': 'C', 'A': '3', 'B': '7',
    'C': '5', 'D': '0', 'E': '6', 'F': '8'
}

# Galois field multiplication for MixColumns
GF4 = {
    '0': '0', '1': '4', '2': '8', '3': 'C',
    '4': '3', '5': '7', '6': 'B', '7': 'F',
    '8': '6', '9': '2', 'A': 'E', 'B': 'A',
    'C': '5', 'D': '1', 'E': '9', 'F': 'D'
}

def gf_multiply(x, y):
    # Convert y to its hexadecimal string representation
    y_hex = hex(y)[2:].upper()

    if x == 1:
        return y
    elif x == 4:
        return int(GF4[y_hex], 16)  # Return the result as an integer
    else:
        return 0


def sub_nibbles(input_block):
    """Substitute each nibble using the S-Box."""
    return ''.join(S_BOX[nibble] for nibble in input_block)

def shift_rows(input_block):
    """Shift rows for a 2x2 matrix representation."""
    return input_block[0] + input_block[3] + input_block[2] + input_block[1]

def mix_columns(input_block):
    s00 = input_block[0]
    s01 = input_block[1]
    s10 = input_block[2]
    s11 = input_block[3]
    r00 = hex(gf_multiply(1, int(s00, 16)) ^ gf_multiply(4, int(s10,
16)))[2:].upper().zfill(1)
```

```python
    r01 = hex(gf_multiply(1, int(s01, 16)) ^ gf_multiply(4, int(s11,
16)))[2:].upper().zfill(1)
    r10 = hex(gf_multiply(4, int(s00, 16)) ^ gf_multiply(1, int(s10,
16)))[2:].upper().zfill(1)
    r11 = hex(gf_multiply(4, int(s01, 16)) ^ gf_multiply(1, int(s11,
16)))[2:].upper().zfill(1)
    return r00 + r01 + r10 + r11

def differential_cryptanalysis(delta_input):
    """Perform differential cryptanalysis."""
    input1 = "1234"  # Some arbitrary input
    input2 = hex(int(input1, 16) ^ int(delta_input,
16))[2:].upper().zfill(4)  # XOR with delta
    output1 = sub_nibbles(shift_rows(mix_columns(input1)))
    output2 = sub_nibbles(shift_rows(mix_columns(input2)))
    delta_output = hex(int(output1, 16) ^ int(output2,
16))[2:].upper().zfill(4)
    return delta_output

# Test
delta_input = "0001"
print(f"Delta Input: {delta_input}")
print(f"Delta Output: {differential_cryptanalysis(delta_input)}")
```



```
PS C:\Users\USER> & C:/Python
보보안/S-AES_diff.py
Delta Input: 0001
Delta Output: 0505
PS C:\Users\USER> & C:/Python
보보안/S-AES_diff.py
Delta Input: 0001
Delta Output: 0505
PS C:\Users\USER>
```

The goal of differential cryptanalysis is to know how the differences in plaintext pairs (known as delta input) propagate through the cipher's operations to produce differences in the resulting ciphertext pairs (known as delta output). By analyzing these relationships, an attacker can infer certain properties about the cipher, potentially revealing weak points or information about the encryption key.

Procedure

1. Choose Delta Input

Select a specific difference (or delta input) between two plaintexts.

Ex)

Let P1 be a plaintext. The chosen delta input ΔP is 0001. Then the second plaintext P2 is P1⊕ ΔP.

2. Propagate Through Cipher

Encrypt both P1 and P2 through 1-round S-AES (without key mixing, as the aim is to analyze the behavior of the cipher's operations). Observe the resulting ciphertexts C1 and C2.

3. Compute Delta Output

Calculate the difference between C1 and C2. This difference is the delta output ΔC=C1⊕C2.

4. Analyze Propagation

In 1-round S-AES, this involves observing how differences propagate through.

SubNibbles: Differential behavior through the S-Box can be analyzed. If certain input differences lead to predictable output differences frequently, then this S-Box can be considered weak for differential attacks.

ShiftRows: Since this operation merely rearranges nibbles, it doesn't introduce new differences but relocates them.

MixColumns: Examine how differences in the input columns transform into differences in the output columns.

5. Record Observations

Keep a record of how different delta inputs affect the resulting delta outputs. If certain patterns are identified, they can be used in more complex differential attacks on multi round versions of the cipher.

6. Deductions

Based on the observations

Identify any weaknesses or patterns in the cipher.

Understand if certain delta inputs lead to certain delta outputs with high probability.

Conclusion

Differential cryptanalysis on 1-round S-AES provides insights into the differential behavior of the cipher's operations. While a single round might not reveal comprehensive weaknesses, understanding the behavior in one round can often be extrapolated to multi-round scenarios, helping cryptanalysts identify potential vulnerabilities or characteristics of the cipher.

References)

Q4) https://jhafranco.com/2012/02/11/simplified-aes-implementation-in-python/

Q5) https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7334988/

https://eprint.iacr.org/2017/1200.pdf