<u>Assignment #2:</u>

Department: Department of Mathematics / College of Natural Science

ID: 20212211

Name: 권대호

Q1. (1 point) 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.

The security strength of a hash function relies on its collision resistance. SHA-256, SHA-384, and SHA-512 algorithms have collision resistances of 128 bits, 192 bits, and 256 bits, respectively. In the case of symmetric algorithms like AES, security strength is directly proportional to the key length. Therefore, AES-128, AES-192, and AES-256 correspond to security strengths of 128 bits, 192 bits, and 256 bits, respectively. This is why we say that SHA-256 matches that of AES-128, SHA-384 matches AES-192, and SHA-512 matches AES-256.[1]

Q2. (1 point) Using Euclid's gcd theorem, determine the following. Show the complete process.

a. gcd(24140,16762)

$$gcd(24140, 16762) = gcd(16762, 24140\%16762) = gcd(16762, 7378)$$

$$= gcd(7378, 16762\%7378) = gcd(7378, 2006)$$

$$= gcd(2006, 7378\%2006) = gcd(2006, 1360)$$

$$= gcd(1360, 2006\%1360) = gcd(1360, 646)$$

$$= gcd(646, 1360\%646) = gcd(646, 68)$$

$$= gcd(68, 646\%68) = gcd(68, 34)$$

$$= gcd(34, 68\%34) = gcd(34, 0)$$

$$\therefore gcd(24140, 16762) = 34$$

Q3. (2 points) 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.

a. 1234 mod 4321

q	r	x	y	a	b	x2	x1	y2	y1
				4321	1234	1	0	0	1
3	619	1	-3	1234	619	0	1	1	-3
1	615	-1	4	619	615	1	-1	-3	4
1	4	2	-7	615	4	-1	2	4	-7
153	3	-307	1075	4	3	2	-307	-7	1075
1	1	309	-1082	3	1	-307	309	1075	-1082
3	0	-1234	4321	1	0	309	-1234	-1082	4321

$$1 = 4321 \times 309 - 1234 \times 1082\ (mod 4321)$$

$$1 = \ 1234 \times (-1082)\ mod 4321 = 1234 \times (3239)\ mod 4321$$

$$\therefore multiplicative\ inverse\ of\ 1234\ mod\ 4321 = 3239$$

[1] *Barker, Elaine (May 2020). "Recommendation for Key Management, Part 1: General", p.54~55*

b. 550 mod 1769

q	r	x	y	a	b	x2	x1	y2	y1
				1769	550	1	0	0	1
3	119	1	-3	550	119	0	1	1	-3
4	74	-4	13	119	74	1	-4	-3	13
1	45	5	-16	74	45	-4	5	13	-16
1	29	-9	29	45	29	5	-9	-16	29
1	16	14	-45	29	16	-9	14	29	-45
1	13	-23	74	16	13	14	-23	-45	74
1	3	37	-119	13	3	-23	37	74	-119
4	1	-171	550	3	1	37	-171	-119	550
3	0	550	-1769	1	0	-171	550	550	-1769

$$1 = -1769 \times 171 + 550 \times 550\ (mod 1769)$$

$$1 = \ 550 \times 550\ mod 1769$$

$$\therefore multiplicative\ inverse\ of\ 550\ mod\ 1769 = 550$$

Q4. (4 points: 2 points for encryption and 2 points for decryption)
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.

a. Encryption

a. Result

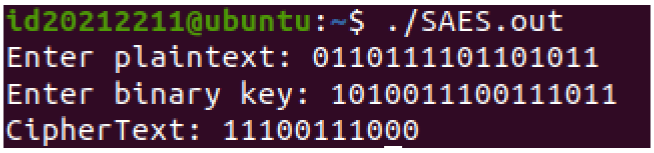
0000111100110000

b. Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

//Binary to Decimal Number
long long BTD(long long binary)
{
    long long num, m = 0, i = 0;
    while (binary > 0)
    {
        num = binary % 10;
        binary = binary / 10;
        m = m + (num << i);
        i++;
    }
    return m;
}

//Decimal to Binary Number
long long DTB(long long decimal)
{
    if (decimal == 0 || decimal == 1)
    {
        return decimal;
    }
    else
    {
        return decimal % 2 + 10 * DTB(decimal / 2);
    }
}
```

```
    //RotNib Function
    long long RotNib(long long rotNibInput)
    {
        long long temp = rotNibInput / 10000;
        rotNibInput = (rotNibInput % 10000) * 10000;
        rotNibInput += temp;

        return rotNibInput;
    }

    //S-Box
    long long S_BOX(long long input)
    {
        int SBOX[16] = {9, 4, 10, 11, 13, 1, 8, 5, 6, 2, 0, 3, 12, 14, 15, 7};

        input = BTD(input);
        input = DTB(SBOX[input]);
        return input;
    }

    //SubNib Function
    long long SubNib(long long subNibInput)
    {
        long long left = subNibInput / 10000;
        long long right = subNibInput % 10000;
        left = S_BOX(left);
        right = S_BOX(right);
        subNibInput = left * 10000 + right;

        return subNibInput;
    }

    int main()
    {
        // Key Generation
        long long plaintext, binaryKey;

        printf("Enter plaintext: ");
        scanf("%lld", &plaintext);


        printf("Enter binary key: ");
        scanf("%lld", &binaryKey);

        long long w0, w1;
        w0 = binaryKey / 100000000;
        w1 = binaryKey % 100000000;

        long long w2 = BTD(w0) ^ BTD(10000000) ^ BTD(SubNib(RotNib(w1)));
        w2 = DTB(w2);

        long long w3 = BTD(w2) ^ BTD(w1);
        w3 = DTB(w3);

        long long w4 = BTD(w2) ^ BTD(110000) ^ BTD(SubNib(RotNib(w3)));
        w4 = DTB(w4);

        long long w5 = BTD(w4) ^ BTD(w3);
        w5 = DTB(w5);

        long long key0 = w0 * 100000000 + w1;
        long long key1 = w2 * 100000000 + w3;
        long long key2 = w4 * 100000000 + w5;

        // Encryption
        //Add Round0 Key
        long long result = BTD(plaintext) ^ BTD(key0);
        result = DTB(result);

        //Round1
        long long resultLeft = result /100000000;
        long long resultRight = result % 100000000;
        resultLeft = SubNib(resultLeft);
        resultRight = SubNib(resultRight);
        result = resultLeft * 100000000 + resultRight;

        long long result0, result1, result2, result3;
        result0 = result / 1000000000000;
        result = result % 1000000000000;
        result1 = result / 100000000;
        result = result % 100000000;
        result2 = result / 10000;
        result = result % 10000;
        result3 = result;

        int M[2][2] = {{1, 2}, {3, 4}};
        int S[2][2] = {{result0, result2}, {result3, result1}};
        int SP[2][2] = {{0, 0}, {0, 0}};

        int b0 = S[0][0] / 1000;
        S[0][0] = S[0][0] % 1000;
        int b1 = S[0][0] / 100;
        S[0][0] = S[0][0] % 100;
        int b2 = S[0][0] / 10;
        S[0][0] = S[0][0] % 10;
        int b3 = S[0][0];

        int b4 = S[1][0] / 1000;
        S[1][0] = S[1][0] % 1000;
        int b5 = S[1][0] / 100;
        S[1][0] = S[1][0] % 100;
        int b6 = S[1][0] / 10;
        S[1][0] = S[1][0] % 10;
        int b7 = S[1][0];

        int c0 = S[0][1] / 1000;
        S[0][1] = S[0][1] % 1000;
        int c1 = S[0][1] / 100;
        S[0][1] = S[0][1] % 100;
        int c2 = S[0][1] / 10;
        S[0][1] = S[0][1] % 10;
        int c3 = S[0][1];

        int c4 = S[1][1] / 1000;
        S[1][1] = S[1][1] % 1000;
        int c5 = S[1][1] / 100;
        S[1][1] = S[1][1] % 100;
        int c6 = S[1][1] / 10;
        S[1][1] = S[1][1] % 10;
        int c7 = S[1][1];

        SP[0][0] = (b0 ^ b6) * 1000 + (b1 ^ b4 ^ b7) * 100 + (b2 ^ b4 ^ b5) * 10 + b3 ^ b5;
        SP[0][1] = (c0 ^ c6) * 1000 + (c1 ^ c4 ^ c7) * 100 + (c2 ^ c4 ^ c5) * 10 + c3 ^ c5;
        SP[1][0] = (b2 ^ b4) * 1000 + (b0 ^ b3 ^ b5) * 100 + (b0 ^ b1 ^ b6) * 10 + b1 ^ b7;
        SP[1][1] = (c2 ^ c4) * 1000 + (c0 ^ c3 ^ c5) * 100 + (c0 ^ c1 ^ c6) * 10 + c1 ^ c7;
        result = (long long)SP[0][0] * 1000000000000 + (long long)SP[1][0] * 100000000 + (long
    long)SP[0][1] * 10000 + (long long)SP[1][1];

        result = BTD(result) ^ BTD(key1);
        result = DTB(result);

        //Final Round
        resultLeft = result / 100000000;
        result = result % 100000000;
        resultRight = result;
        resultLeft = SubNib(resultLeft);
        resultRight = SubNib(resultRight);
        result = resultLeft * 100000000 + resultRight;
        result0 = result/1000000000000;
        result = result%1000000000000;
        result1 = result/100000000;
        result = result%100000000;
        result2 = result/10000;
        result = result%10000;
        result3 = result;
        result = result0*1000000000000 + result3*100000000 + result2*10000 + result1;
    result = BTD(result) ^ BTD(key2);
    result = DTB(result);

    printf("CipherText: %lld\n", result);

    return 0;
}
```

c. Execution proof



```
id20212211@ubuntu:~$ ./SAES.out
Enter plaintext: 0110111101101011
Enter binary key: 1010011100111011
CipherText: 11100111000
```

b. Decryption

a. Result

0110111101101011

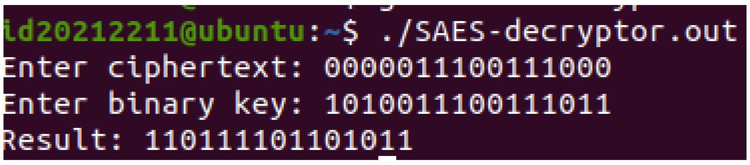
b. Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

//Binary to Decimal Number
long long BTD(long long binary)
{
    long long num, m = 0, i = 0;
    while (binary > 0)
    {
        num = binary % 10;
        binary = binary / 10;
        m = m + (num << i);
        i++;
    }
    return m;
}

//Decimal to Binary Number
long long DTB(long long decimal)
{
    if (decimal == 0 || decimal == 1)
    {
        return decimal;
    }
    else
    {
        return decimal % 2 + 10 * DTB(decimal / 2);
    }
}

//RotNib Function
long long RotNib(long long rotNibInput)
{
    long long temp = rotNibInput / 10000;
    rotNibInput = (rotNibInput % 10000) * 10000;
    rotNibInput += temp;

    return rotNibInput;
}

//invS-Box
long long S_BOX(long long input)
{
    int invSBOX[16] = {10, 5, 9, 11, 1,7, 8, 15, 6, 0, 2, 3, 12, 4, 13, 14};

    input = BTD(input);
```

```c
        input = DTB(invSBOX[input]);
        return input;
    }
    long long prev_S_BOX(long long input)
    {
        int SBOX[16] = {9,4,10,11,13,1,8,5,6,2,0,3,12,14,15,7};

        input = BTD(input);
        input = DTB(SBOX[input]);
        return input;
    }

    //SubNib Function
    long long SubNib(long long subNibInput)
    {
        long long left = subNibInput / 10000;
        long long right = subNibInput % 10000;
        left = S_BOX(left);
        right = S_BOX(right);
        subNibInput = left * 10000 + right;

        return subNibInput;
    }

    long long prev_SubNib(long long subNibInput)
    {
        long long left = subNibInput / 10000;
        long long right = subNibInput % 10000;
        left = prev_S_BOX(left);
        right = prev_S_BOX(right);
        subNibInput = left * 10000 + right;

        return subNibInput;
    }

    int main()
    {
        // Key Generation
        long long ciphertext, binaryKey;

        printf("Enter ciphertext: ");
        scanf("%lld", &ciphertext);


        printf("Enter binary key: ");
        scanf("%lld", &binaryKey);

        long long w0, w1;
        w0 = binaryKey / 100000000;
        w1 = binaryKey % 100000000;

        long long w2 = BTD(w0) ^ BTD(10000000) ^ BTD(prev_SubNib(RotNib(w1)));
        w2 = DTB(w2);

        long long w3 = BTD(w2) ^ BTD(w1);
        w3 = DTB(w3);

        long long w4 = BTD(w2) ^ BTD(110000) ^ BTD(prev_SubNib(RotNib(w3)));
        w4 = DTB(w4);

        long long w5 = BTD(w4) ^ BTD(w3);
        w5 = DTB(w5);

        long long key0 = w0 * 100000000 + w1;
        long long key1 = w2 * 100000000 + w3;
        long long key2 = w4 * 100000000 + w5;


        long long result = BTD(ciphertext)^BTD(key2);
        long long result0, result1, result2, result3;
        result = DTB(result);
        result0 = result/1000000000000;
        result = result%1000000000000;
    result1 = result/100000000;
    result = result%100000000;
    result2 = result/10000;
    result = result%10000;
    result3 = result;
    result = result0*1000000000000 + result3*100000000 + result2*10000 + result1;

    long long resultLeft = result/100000000;
    long long resultRight = result%100000000;
    resultLeft = SubNib(resultLeft);
    resultRight = SubNib(resultRight);
    result = resultLeft*100000000+resultRight;
    result = BTD(result)^BTD(key1);
    result = DTB(result);


    result0 = result / 1000000000000;
    result = result % 1000000000000;
    result1 = result / 100000000;
    result = result % 100000000;
    result2 = result / 10000;
    result = result % 10000;
    result3 = result;

    int S[2][2] = {{result0, result2},{result1, result3}};
    int SP[2][2] = {{0,0},{0,0}};
    int b0 = S[0][0] / 1000;
    S[0][0] = S[0][0] % 1000;
    int b1 = S[0][0] / 100;
    S[0][0] = S[0][0] % 100;
    int b2 = S[0][0] / 10;
    S[0][0] = S[0][0] % 10;
    int b3 = S[0][0];

    int b4 = S[1][0] / 1000;
    S[1][0] = S[1][0] % 1000;
    int b5 = S[1][0] / 100;
    S[1][0] = S[1][0] % 100;
    int b6 = S[1][0] / 10;
    S[1][0] = S[1][0] % 10;
    int b7 = S[1][0];

    int c0 = S[0][1] / 1000;
    S[0][1] = S[0][1] % 1000;
    int c1 = S[0][1] / 100;
    S[0][1] = S[0][1] % 100;
    int c2 = S[0][1] / 10;
    S[0][1] = S[0][1] % 10;
    int c3 = S[0][1];

    int c4 = S[1][1] / 1000;
    S[1][1] = S[1][1] % 1000;
    int c5 = S[1][1] / 100;
    S[1][1] = S[1][1] % 100;
    int c6 = S[1][1] / 10;
    S[1][1] = S[1][1] % 10;
    int c7 = S[1][1];

    SP[0][0] = (b3^b5)*1000 + (b0^b6)*100 + (b1^b4^b7)*10 + (b2^b4^b3);
    SP[1][0] = (b1^b7)*1000 + (b2^b4)*100 + (b0^b3^b5)*10 + (b0^b6^b7);
    SP[0][1] = (c3^c5)*1000 + (c0^c6)*100 + (c1^c4^c7)*10 + (c2^c4^c3);
    SP[1][1] = (c1^c7)*1000 + (c2^c4)*100 + (c0^c3^c5)*10 + (c0^c6^c7);


    result = (long long)SP[0][0]*1000000000000 + (long long)SP[1][1]*100000000 + (long
long)SP[0][1]*10000 + (long long)SP[1][0];

    resultLeft = result/100000000;
    result = result%100000000;
    resultRight = result;
    resultLeft = SubNib(resultLeft);
    resultRight = SubNib(resultRight);
    result = resultLeft*100000000 + resultRight;
```

```
        result = BTD(result)^BTD(key0);
        result = DTB(result);
        printf("Result: %lld\n", result);


        return 0;
    }
```

c. Execution Proof



```
id20212211@ubuntu:~$ ./SAES-decryptor.out
Enter ciphertext: 0000011100111000
Enter binary key: 1010011100111011
Result: 1101111011010011
```

Q5. (2 points) Implement a differential cryptanalysis attack on 1-round S-AES. Explain the logic behind the attack. Show all steps.

Select a random round key, and generate two plaintexts that have a similar structure with minimal bit differences between them. First, encrypt the two plaintexts. Calculate the bit difference between the two ciphertexts by performing an XOR operation. Then, pass this difference through the inverse of the S-Box to convert it into a difference between the plaintexts.

In this way, crackers can discover the differential characteristic of the round key. Once he has identified the differential characteristic, he can make educated guesses about the possible values of the round key and iterate through the previous steps. Continue this process to solve the problem.