COMPSCI 4CR3

Assignment 2 — Dev Mody

Student Number: 400445340
Date: Saturday October 12th 2024
Macid: modyd1

Question 1:

1. An encryption function $e(k, x)$, where k is the key and x is the message, is said to be linear if $e(k, x + y) = e(k, x) + e(k, y)$ for all keys k and all messages x, y.

   (a) (15 points) Explain why a secure encryption function must not be linear; provide an example scenario where Trudy could dangerously exploit this linearity.

   (b) (15 points) Recall that the S-box in AES is a function

   $$f : \{0, 1, \ldots, 255\} \to \{0, 1, \ldots, 255\}.$$

   The actual S-box is given by the following table.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	1	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	4	C7	23	C3	18	96	5	9A	7	12	80	E2	EB	27	B2	75
4	9	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	0	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	2	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	6	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	8
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	3	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

   For an input (uv), in base 16, the output is located in row u and column v. For example, $f(1E) = 72$. Show that this S-box is not a linear function.

a) Say we have an encryption function e s.t. $e(k,x)$ uses a key k to encrypt plaintext x to derive some ciphertext. The function e is linear if $\forall x,y\ \forall k\ (e(k,x+y) = e(k,x) + e(k,y))$. Secure encryption functions should not be linear because linearity makes the security more vulnerable to various forms of attacks, especially analytical ones. For instance, the property $e(k,x+y) = e(k,x)+e(k,y)$ could be exploited by attackers who can intercept or generate ciphertexts from known plaintext. Here's an example of a way Trudy could exploit this linearity:

Assume he doesn't know the plaintexts x and y. However, since $e(k,x)$ and $e(k,y)$ are public, he knows $e(k,x+y)$ without even knowing plaintext $x+y$. A technique he could use is from the known plaintext attack where Trudy knows (either of the key or stream) and the ciphertext. Then he can generate a system of linear equations where the feedback coefficients are the unknowns to be determined. Even without sophisticated techniques, Trudy could just continue observe various messages $x_1, \ldots, x_n$ to start seeing linear combinations to gain structural information regarding the encryption scheme. This is a security issue because linearity undermines confusion in the way that the relationship between plaintext, ciphertext & key is simple enough for the attacker to predict the output. In addition, it undermines diffusion since changes in plaintext make predictable changes to ciphertext.

b) Proof By Contradiction: Assume that the AES S-box is linear. Then we want to show that the property $f(x \oplus y) = f(x) \oplus f(y)$ for $f: \mathbb{Z}_{256} \to \mathbb{Z}_{256}$ and any inputs $x = ab$ and $y = cd$ where $a,b,c,d \in \{0,1,2,\ldots,9,a,b,\ldots,F\}$. The reason we choose to use the property $f(x \oplus y) = f(x) \oplus f(y)$ is because as we are adding in Extension Fields for AES, we use Definition 4.3.3 where $A(x)+B(x) = \sum_{i=0}^{m-1} c_i x^i$ where $c_i \equiv a_i + b_i \bmod 2 \equiv a_i \text{ XOR } b_i$ for any $A(x), B(x) \in GF(2^n)$ to show we must show $f(x \oplus y) = f(x) \oplus f(y)$.

Let's say $x = 1E$ and $y = 5A$. This means the output of $f(x)$ is in row 1 and column E of the S-box $[f(x) = 72]$ and similarly for $f(y)$, $f(y) = BE$. To determine $x + y$ and $f(x) + f(y)$, we can convert both operands into binary values and perform binary addition and convert back to hex.

$$x \oplus y = (1E)_{hex} \oplus (5A)_{hex}$$
$$= (00011110)_2 \oplus (01011010)_2$$
$$= (01000100)_2$$
$$= (44)_{hex}$$
$$f(x \oplus y) = (1B)_{hex}$$

$$f(x) \oplus f(y) = (72)_{hex} \oplus (BE)_{hex}$$
$$= (01110010)_2 \oplus (10111110)_2$$
$$= (11001100)_2$$
$$= (CC)_{hex}$$

$(1B)_{hex} \neq (CC)_{hex}$

As a result, since $f(x) \oplus f(y) \neq f(x \oplus y)$ we can say that we have reached a contradiction as the property of linearity does not hold for $f: \mathbb{Z}_{256} \to \mathbb{Z}_{256}$.

Question 2:

2. (45 points) Suppose we replace the functions in SHA-256 with following functions.

$$\mathrm{Ch}(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$$
$$\mathrm{Ma}(x, y, z) = (x \wedge y) \oplus (x \wedge \neg z) \oplus (y \wedge z)$$
$$\Sigma_0(x) = (x \gg_R 2) \oplus (x \gg_R 23) \oplus (x \gg_R 12)$$
$$\Sigma_1(x) = (x \gg_R 16) \oplus (x \gg_R 21) \oplus (x \gg_R 15)$$
$$\sigma_0(x) = (x \gg_R 17) \oplus (x \gg_R 11) \oplus (x \gg_S 13)$$
$$\sigma_1(x) = (x \gg_R 7) \oplus (x \gg_R 9) \oplus (x \gg_S 12)$$

Compute the hash of the following message using the modified SHA-256:
`The quick brown fox jumped over the lazy dog.`
Rules to follow:

- The input to the hash should be a byte string. For example, in Python, add a `b` so that the input looks like `b'The quick brown fox jumped over the lazy dog.'`
- The output should be a list of bytes. For the hash of the above message, represent the output bytes in base 16 as a string. Example input and output:
  COMPSCI 4CR3
  8b709893e1b5f6008bfa29295ab4dd2fc0cc81cb68e22c5bb6a69a79d57e6db4
- Include your source code; Python is preferred, but you can use other languages.

Code Implementation Found in question2.py in the answers submitted:

```
def CH(x, y, z):
    return (x & y) ^ (x & z)

def MA(x, y, z):
    return (x & y) ^ (x & ~z) ^ (y & z)

def sigma0(x):
    return (ROTR(x, 2) ^ ROTR(x, 23) ^ ROTR(x, 12))

def sigma1(x):
    return (ROTR(x, 16) ^ ROTR(x, 21) ^ ROTR(x, 15))

def sigma_0(x):
    return (ROTR(x, 17) ^ ROTR(x, 11) ^ (SHR(x, 13)))

def sigma_1(x):
    return (ROTR(x, 7) ^ ROTR(x, 9) ^ (SHR(x,12)))

def ROTR(x, n):
    return (x >> n) | (x << (32 - n)) & 0xFFFFFFFF

def SHR(x, n):
    return x >> n
```

```
K = [0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
     0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
     0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
     0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
     0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
     0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
     0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
     0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2]

H = [0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a, 0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19]
```

```
def modified_sha256(message : str):

    def message_pad(message):
        # Convert the message to bytes using UTF-8 encoding
        message = message.encode('utf-8')

        # Convert each byte to a binary string
        bit_string = ''.join(format(byte, '08b') for byte in message)
        initial_pad = bit_string + '1'

        # Padding to 448 bits mod 512
        k = (448 - (len(initial_pad) % 512)) % 512
        append_zeros = "0" * k

        # Append the original length as a 64-bit integer
        padding_64 = format(len(bit_string), '064b')

        # Final padded message: initial + zeros + length
        final_padding = initial_pad + append_zeros + padding_64
        return [final_padding[i:i+512] for i in range(0, len(final_padding), 512)]

    def message_sched(block):
        W = [0] * 64
        for j in range(16):
            W[j] = int(block[j*32:(j+1)*32], 2)
        for j in range(16, 64):
            W[j] = (sigma_1(W[j-2]) + W[j-7] + sigma_0(W[j-15]) + W[j-16]) & 0xFFFFFFFF
        return W

    def compression(h, W):
        A, B, C, D, E, F, G, HH = h
        for j in range(64):
            T1 = (HH + sigma1(E) + CH(E, F, G) + K[j] + W[j]) & 0xFFFFFFFF
            T2 = (sigma0(A) + MA(A, B, C)) & 0xFFFFFFFF
            HH = G
            G = F
            F = E
            E = (D + T1) & 0xFFFFFFFF
            D = C
            C = B
            B = A
            A = (T1 + T2) & 0xFFFFFFFF
        return [(h[i] + val) & 0xFFFFFFFF for i, val in enumerate([A, B, C, D, E, F, G, HH])]

    blocks = message_pad(message)

    h = H.copy()

    for block in blocks:
        w = message_sched(block)
        h = compression(h, w)

    return ''.join(format(word, '08x') for word in h)
```

My Output For b'The quick brown fox jumped over the lazy dog':

a909986f702bf8d63dd741ff69ebb34c6ed957da07a9609322e43791edb7746b

Question 3:

3. (25 points) Let $h_1 : \{0,1\}^* \to \{0,1\}^n$ and $h_2 : \{0,1\}^* \to \{0,1\}^n$ be hash functions with output lengths of n bits. Construct a hash function $h : \{0,1\}^* \to \{0,1\}^{2n}$ by concatenating the outputs of h_1 and h_2, that is, $h(x) = h_1(x)||h_2(x)$ where $||$ is concatenation. Suppose h_1 is collision resistant but h_2 is not. Can we say that h is collision resistant? Prove your claim.

Suppose we have a collision resistant hash function $h_1 : \{0,1\}^* \to \{0,1\}^n$ and a non collision-resistant hash function $h_2 : \{0,1\}^* \to \{0,1\}^n$ where the output of both f^n is fixed of n bits. Say we construct another hash function, $h(x) = h_1(x) || h_2(x)$. Then we still know it is collision-resistant.

Proof By Contradiction. Assume that $h(x) = h_1(x) || h_2(x)$ is not collision resistant for any collision resistant h_1 and non-collision resistant h_2. Then we must show that is computationally feasible to find some $x_1, x_2 \in \{0,1\}^*$ s.t. $x_1 \neq x_2$ and $h(x_1) = h_1(x_1) || h_2(x_1) = h_1(x_2) || h_2(x_2) = h(x_2)$. To start, we know that h_2 is not collision-resistant. Then it is computationally feasible to find some $x_1, x_2 \in \{0,1\}^*$ s.t. $x_1 \neq x_2$ and $h_2(x_1) = h_2(x_2)$. Then we can say it is computationally feasible to find some $x_1, x_2 \in \{0,1\}^+$ s.t. $x_1 \neq x_2$ and $h_1(x_1) || h_2(x_1) = h_2(x_2) || h_2(x_2)$. However when observe that for the same $x_1 \neq x_2$ and the fact that h_1 is collision resistant meaning it is computationally infeasible to find some $x_1, x_2 \in \{0,1\}^*$ s.t. $x_1 \neq x_2$ and $h_1(x_1) = h_1(x_2)$. We can say that even if $h_2(x_1) = h_2(x_2)$, $h_1(x_1) || h_2(x_1) \neq h_1(x_2) || h_2(x_2)$. This can be seen as we can cancel the concatenation of $h_2(x_1)$ and $h_2(x_2)$ on both sides of the equality to get the equality $h_1(x_1) = h_1(x_2)$ which is computationally infeasible to derive such x_1 & x_2 for. As a result, we've reached a contradictory statement as h_1 is known to be collision resistant $\therefore$ Hence proven.