Lab 3 Solutions

Cody Lewis

August 23, 2019

Part 1

1

A function that is easy to compute one way, but hard to compute the other way.

Finding $H(x)$ given x	Easy
Finding x given $H(x)$	Hard

2

Same as a one way function, only there exists a trapdoor which when known makes reversing the function easy to compute.

3

Key Generation

Choose 2 large primes p and q and compute,

$$n = pq \tag{1}$$
$$m = \phi(n) = \phi(pq) = (p-1)(q-1) \tag{2}$$

Choose $e \in [1, m-1]$, such that $gcd(e, m) = 1$

Find d such that $ed = 1 \bmod m$

Encryption

$$M^e \bmod n \tag{3}$$

Decryption

$$C^d \bmod n = M^{d \cdot e} \bmod n = M \tag{4}$$

4

A way of providing a message integrity check using a shared key, authentication is also provided on this message due to that usage of the shared key.

5

- Easy to generate
- Easy to verify
- Unforgeability
- Non-repudiation

Part 2

6

a

The mac system is a pair of algorithms, the Mac which generates a tag, t, and the verify which checks whether a given tag matches the given message.

$$Mac(k, m) \to t \tag{5}$$
$$Ver(k, m', t) \to \{True, False\} = (Mac(k, m') \equiv t) \tag{6}$$

b

The double hash mitigates the ability to perform an insertion attack, or a collision.

The insertion attack could occur if $HMAC(k, m) = H(k||m)$, an attacker could insert more data into the message until the same tag is generated from the $HMAC$.

The collision attack could occur if $HMAC(k, m) = H(m||k)$, an attacker could find collision without need knowledge of the key.

c

MAC uses symmetric keys, yet digital signatures use public key cryptography. While both provide integrity to the message, digital signatures also provide authentication for the sender. However, since MACs use symmetric key cryptography, they are much faster and more efficient than digital signatures.

d

Digital signatures can replace MACs in terms of functionalities. However, MAC have much faster computability as they use symmetric key cryptography rather than public key cryptography which is used for digital signatures.

7

a

$$A \to B : E_k(m||H(m)) \tag{7}$$

This assumes that A and B have already agreed on a symmetric key, k.

b

$$A \to B : E_{Puk_B}(m||E_{Prk_A}(m)) \tag{8}$$

c

$$A \to B : E_{Puk_B}(K||E_{Prk_A}(A)) \tag{9}$$
$$A \to B : E_K(m||HMAC(K, m)) \tag{10}$$

d

Encrypt than sign means that the receiver would need to keep a copy of the cipher-text in order to verify integrity, whereas sign then encrypt means they would only need to keep the plain-text.

8

Below is a python implementation of the CFB cipher,

```
#!/usr/bin/env python3

'''
A simple CFB moded block cipher.

Author: Cody Lewis
Date: 2019-08-16
'''

def encipher(text, key):
    '''The cipher block.'''
    return bin(text ^ key)[2:]

def xor(a, b):
    '''XOR 2 binary strings.'''
    result = ""
    for i, j in zip(a, b):
        if i == j:
            result += "0"
        else:
            result += "1"
    return result

def encrypt(plaintext, key, IV, k):
    '''Encrypt using CFB mode with self-syncronous shift.'''
    cur_cipher = ""
    ciphertext = ""
    shift = bin(IV)[2:]
    for i in range(0, len(plaintext), k):
        cur_cipher = xor(encipher(int(shift, 2), key)[:k], plaintext[i:i + k])
        ciphertext += cur_cipher
        shift = bin(((int(shift, 2) << k) + int(cur_cipher, 2)) % (2**16))[2:]
    return process_output(ciphertext)

def decrypt(ciphertext, key, IV, k):
    '''Decrypt using CFB mode with self-syncronous shift.'''
    plaintext = ""
    shift = bin(IV)[2:]
    for i in range(0, len(ciphertext), k):
        plaintext += xor(encipher(int(shift, 2), key)[:k], ciphertext[i:i + k])
        shift = bin(
            ((int(shift, 2) << k) + int(ciphertext[i:i + k], 2)) % (2**16)
        )[2:]
    return process_output(plaintext)


def process_input(text):
    '''Make the input suitable for the encrypt/decrypt.'''
    bin_text = bin(int(text, 16))[2:]
    return bin_text.zfill(len(bin_text) + ((16 - len(bin_text)) % 16))

def process_output(text):
    '''Make the output suitable for printing.'''
    return hex(int(text, 2))[2:]

if __name__ == '__main__':
    IV = input("Input your IV: ")
    OG_IV = IV
    IV = int(IV[:4], 16)
    KEY = input("Input your key: ")
    OG_KEY = KEY
    KEY = int(KEY[:4], 16)
    PLAINTEXT = input("Input your plaintext: ")
    K = int(input("Input your shift bits: "))
    print()
    print(f"IV:\t\t{OG_IV}")
    print(f"Key:\t\t{OG_KEY}")
    CIPHERTEXT = encrypt(process_input(PLAINTEXT), KEY, IV, K)
    print(f"Ciphertext:\t{CIPHERTEXT}")
    print(f"Plaintext:\t{decrypt(process_input(CIPHERTEXT), KEY, IV, K)}")
```

Part 3

9

$$19 \bmod 13 \tag{11}$$

Since $19 \geq 13$, we can bring it down into the range of the modulus, this can be done by looking at the new value from the first step of the Euclidean algorithm,

$$\begin{aligned} \gcd(13, 19) &: 19 = 13 + 6 \\ \therefore 19 &\bmod 13 = 6 \end{aligned}$$

$$4 \bmod 13 \tag{12}$$

Since $0 \leq 4 < 13$,

$$4 \bmod 13 = 4 \tag{13}$$

$$\begin{aligned} \gcd(3, 13) :& 13 = 3 \cdot 4 + 1 \\ \gcd(1, 3) :& 3 = 3 \cdot 1 \\ \therefore \gcd(3, 13) &= 1 \end{aligned}$$

$$\begin{aligned} \gcd(4, 42) :& 24 = 10 \cdot 4 + 2 \\ \gcd(2, 4) :& 4 = 4 \cdot 2 \\ \therefore \gcd(4, 42) &= 2 \end{aligned}$$

The following equation may be formed,

$$11b \bmod 24 = 1. \tag{14}$$

Then the extended Euclidean algorithm may be applied to find b,

first find greatest common divisors until there is a 1 remainder,

$$\begin{aligned}\gcd(11,24)&:24 = 2\cdot 11 + 2\\ \gcd(2,11)&:11 = 2\cdot 5 + 1\end{aligned}$$

then solve for one,

$$\begin{aligned}1 &= 11 - 2\cdot 5\\ &= 11 - (24 - 2\cdot 11)\cdot 5 && \text{Substituting 2 for the rearranged equation formed by } \gcd(11,24)\\ &= 11 - 24 + 10\cdot 11\\ &= 11\cdot 11 - 24 && \text{This shows that 11 multiplied by 11 gives 1}\\ \therefore b &= 11.\end{aligned}$$