CSCI971 Advance Computer Security: Homework #1

Mei Wangzhihui 2019124044

Problem 1

Solution

1. G' is composing PRG so G' is secure.

2. As algorithm G is an efficient deterministic algorithm, so the $msb(G(0))$ is either 0 or 1, we assume that $msb(G(0)) = 1)$.

We define a statistical test $A(x)$ as :

if $[msb(x)]$ output "1" else output 0

As the G is a secure PRG, so the probability of the

$$\begin{aligned} Adv_{PRG}[A, G] &= |Pr(A(G(0) = 1) - Pr(A(r) = 1)| \\ &= 1/2 \end{aligned}$$

It is not negligible. So $G'(k)$ is not secure PRG.

3. As $G(k)$ is secure PRG, the $G'(k)$ is secure PRG.

4. We can find that:

$$k \oplus 1^s = \bar{k}$$

because $k \leftarrow \{0,1\}^s$ so $\bar{k} \leftarrow \{0,1\}^s$, thus:

$$\begin{aligned} Adv_{PRG}[A, G'] &= |Pr(A(G(\bar{k}) = 1) - Pr(A(r) = 1)| \\ &= |Pr(A(G(k) = 1) - Pr(A(r) = 1)| \end{aligned}$$

G is secure so G' is secure.

5.

$$\begin{aligned} Adv_{PRG}[A, G'] &= |Pr(A(reserver(G(k)) = 1) - Pr(A(r) = 1)| \\ &= |Pr(A(G(k) = 1) - Pr(A(r) = 1)| \end{aligned}$$

G is secure so G' is secure.

Problem 2

Solution

$$\begin{aligned} Pr(lsb(G'(k_1, k_2)) = 1) &= Pr(lsb(G(k_1)) = 1) \times Pr(lsb(G(k_2)) = 1) \\ &= 1/4 \end{aligned}$$

so

$$\begin{aligned} Adv_{PRG}[A, G'] &= |Pr(A((lsb(G'(k_1, k_2)) = 1) - Pr(A(r) = 1)| \\ &= 1/4 \end{aligned}$$

Problem 3

Solution

1. The $E'\left((k, k'), m\right) = E(k, m) \| E\left(k', m\right)$ assume an Attacker game. We have:

$$Adv_{SS}(A, E') = 2Adv_{SS}(A.E)$$

E' is semantically secure.

Problem 3 continued on next page...

2. As the k is fixed , the c_1 and c_2 is fixed too. The $Adv_{SS} = 1$ So it is not semantically secure.
3. The k from the $E_{\prime}$ leak the key. The adversary can encrypt the m_0 m_1 to get the ciphertext c_0 c_1, then the $Adv_{SS} = 1$ So it is not semantically secure.
4. A can deduce b from the $LSB(m)$ part of $E'(k, m)$, if $LSB(m_0) \neq LSB(m_1)$ the Adv_{SS} is not negligible,So it is not semantically secure.

Problem 4

Solution We can know that:

$$c_1 = m_1 \oplus k$$

then:

e

$$k = c_1 \oplus m_1$$

$so:$

$$\begin{aligned} c_2 &= m_2 \oplus k \\ &= m2 \oplus c_1 \oplus m1 \\ &= 6c73d5240a948c86981bc2808548 \end{aligned}$$

the program code are here:

```
from functools import reduce


def ByteToHex(bins):
    return ''.join(["%02x" % x for x in bins]).strip()


def BytesXor(b1, b2):
    return [a ^ b for a, b in zip(b1, b2)]


m1 = "attack at dawn".encode('ascii')
m2 = "attack at dusk".encode('ascii')
c1 = bytes.fromhex("6c73d5240a948c86981bc294814d")
c2 = reduce(BytesXor, [m1, m2, c1])
print(ByteToHex(c2))
```

Problem 5

Solution We can know that:

$$c_x \oplus c_y = m_x \oplus m_y$$

and another fact: the ASCII of space is 32 so for the plaintext:

$$X \oplus 32 = x$$

$$x \oplus 32 = X$$

x denote the lower letter whose ASCII is between 97-122 and X denote upper letter whose ASCII is between 65-90.

we can conclude that if two A xor B is still letter, then either A or B is space. That is:

$$A \oplus B = C, C \in Z_{space} \Rightarrow \text{either A or B is space}$$

so, first we can find certain spaces from point 2.
for the jth char of ith ciphertext $C_{i,j}$, $\forall j \in [1, 10]$ and $j \neq i$, if $C_{i,t} \oplus C_{j,t} = X$, if X is letter then $C_{i,t}$ is space.
then from point 1 we can get K_{space}

$$K_{space} = C_{space} \oplus P_{space}$$

then we can use the K_{space} to decrypt the letter in ciphertext in the same position with space. Then we can guess the plaintext P_{part} from the text. and find the K_{rest}. Loop until find all key K_{all}.
Finally, we can use the K_{all} to decrypt the target ciphertext C_{target} the program code are here:

```
import collections
import string


# XORs two string
def str_xor(a, b):  # xor two strings (trims the longer input)
   return "".join([chr(ord(x) ^ ord(y)) for (x, y) in zip(a, b)])

# Initialize the ciphertexts list and targetciphertext
def read_ciphertexts(ciphertextpath, targetciphertextpath):
   cyphertext_list = []
   target_ciphertext = ""
   with open(ciphertextpath) as f:
      lines = f.readlines()
      for line in lines:
         cyphertext_list.append(line.strip('\n'))
   with open(targetciphertextpath) as f:
      target_ciphertext = f.readline().strip('\n')
   return cyphertext_list, target_ciphertext

# Gusee key by the space
def dectect_key(ciphertext_list):
   # For each ciphertext
   final_key = [None]*150
   possible_space_idxs = []
   for i, cti in enumerate(ciphertext_list):
      counter = collections.Counter()
      for j, ctj in enumerate(ciphertext_list):
         if i != j: # Just dont compare with itself
            for k, char in enumerate(str_xor(cti, ctj)):
               if char in string.printable and char.isalpha():
                  # space(0x20) ^ letter == letter
                  # the kth position are likely to be the space
                  counter[k] += 1

      for i, val in list(counter.items()):
         # assume position with this situation occuring no less than 6 times as space.
         if val >= 6:
            possible_space_idxs.append(i)

      # This is core idea: XOR the current ciphertext with spaces, we can get key in these
          positions.
```

```
        space_xor_test = str_xor(cti, ' '*150)
        for i in possible_space_idxs:
            final_key[i] = space_xor_test[i]
            possible_space_idxs.add(i)
    return final_key, possible_space_idxs


def run(target):
    ciphertext_list, target_ciphertext = read_ciphertexts(
        "./ciphertext.txt", "./target_ciphertext.txt")
    final_key,possible_space_idxs = dectect_key(ciphertext_list)
    final_key_hex = ''.join(
        [val if val is not None else '00' for val in final_key])
    output = str_xor(target_ciphertext, final_key_hex)
    print(''.join(
        [char if index in possible_space_idxs else '*' for index, char in enumerate(output)]))
    print(str_xor(final_key_hex, target_ciphertext))
if __name__ == "__main__":
    run()
```

We get the final Key:
K =66396e89c9dbd8cc9874352acd6395102eafce78aa7fed28a0
7f6bc98d29c5 0b69b0339a19f8aa401a9c6d708f80c066c763fef
0123148cdd8e802d05ba98777335daefcecd59c433a6b268b60bf4
ef03c9a611098bb3e9a3161edc7b804a33522cfd202d2c68c57376
edba8c2ca50027c61246e2a12b0c4502175010c0a1ba4625786d91
1100797d8a47e98b0204c4ef06c867a950f11ac989dea88fd1dbf1
6748749ed4c6f45b34c9d96c4
M_{target} = When using a stream cipher, never use the key more than once