Cryptography module, Exercise 2 - Answers

All the code contained in this exercise solution is hosted at: https://github.com/amar-laksh/UNI/assignments/CRYPTO/code/ex2

Answer - 1:

As given in the given question we assume:
k0 = 00000000000000000000000000000000
k1 = 00000000000000000000000000000000
plaintext = 00000000000000000000000000000000

Listing 1 contains the python code and **Listing 2** contains the output.

Listing 1: ex2.1.py

```
import sys
sys.path.insert(1, 'Python-AES')
from AES import *

plaintext = "00000000000000000000000000000000"
key = "00000000000000000000000000000000"
key = process_key(key, 4)
Nr = 0
expanded_key = KeyExpansion(key, 4, 4, Nr)
block = process_key(plaintext)
block = Cipher(block, expanded_key, 4, 4, Nr)
print("PLAINTEXT:\t {0}".format(plaintext))
print("ENCRYPTED:\t {0}".format(str_block_line(block)))

block = InvCipher(block, key, 4, 4, Nr)
print("DECRYPTED:\t {0}".format(str_block_line(block)))
```

Listing 2: output of ex2.1.py

```
PLAINTEXT:      00000000000000000000000000000000
ENCRYPTED:      63636363636363636363636363636363
```

Answer - 2:

As given in the given question we assume:
plaintext = 00000000000000000000000000000000
key = ffffffffffffffffffffffffffffffff

Note we set the key as all *Fs* and not *1s* because the key is supposed to be a bit string of 1s.

Listing 3 contains the python code and **Listing 4** contains the output.

Listing 3: ex2.2.py

```
import sys
sys.path.insert(1, 'Python-AES')
from AES import *

plaintext = "00000000000000000000000000000000"
key = "ffffffffffffffffffffffffffffffff"
key = process_key(key, 4)
Nr = 1
expanded_key = KeyExpansion(key, 4, 4, Nr)


block = process_key(plaintext)

#  Printing out subkeys
print("key:\t {0}".format(str_block_line(expanded_key)))
c = 0
for i in range(0, (32 * Nr) + 1, 32):
    print("k{}:\t{}".format(c, str_block_line(expanded_key)[i:i+32]))
    c += 1
print("PLAINTEXT:\t {0}".format(plaintext))


block = Cipher(block, expanded_key, 4, 4, Nr)
print("ENCRYPTED:\t {0}".format(str_block_line(block)))

block = InvCipher(block, expanded_key, 4, 4, Nr)
print("DECRYPTED:\t {0}".format(str_block_line(block)))
```

Listing 4: output of ex2.2.py

```
key:    ffffffffffffffffffffffffffffffffe8e9e9e917161616e8e9e9e917161616
k0:     ffffffffffffffffffffffffffffffff
k1:     e8e9e9e917161616e8e9e9e917161616
PLAINTEXT:          00000000000000000000000000000000
ENCRYPTED:          fefffff01000000fefffff01000000
DECRYPTED:          00000000000000000000000000000000
```

Answer - 3:

The first thing to notice about this question is to recognize what property of the Hash function we are trying to attack.

Here we have two options according to the information given in the question:

- **Second pre-image resistance(Henceforth called Sec):** By finding a hash *h'* such that given message *M* and hash function *f*, we have: *f(M) = h = h'*.

  According to the question we can try to find a hash that satisfies they breaks the Sec property given message = "mark" but since the hash function **MY60SHA** is just a shortened version of the 128 bit **SHA-1** function this kind of attack is computationally infeasible.

- **Collision resistance(Henceforth called Col):** By finding two different messages *M* and *M'* that have the same hash *f(M) = f(M')*. This is often the first property to be broken. Hence we should focus our efforts onto this property.

There are two approaches to breaking the *Col* property:

- **Birthday Paradox Approach:** The birthday paradox is the counter-intuitive principle that for groups of as few as 23 persons there is already a chance of about one half of finding two persons with the same birthday (assuming all birthdays are equally likely and disregarding leap years). Compared to finding someone in this group with your birthday where you have 23 independent chances and thus a success probability of $\frac{23}{365} \approx 0.06$, this principle is based on the fact that there are $23 \times 222 = 253$ distinct pairs of persons. This leads to a success probability of about 0.5(note that this does not equal $\frac{253}{365} \approx .7$ since these pairs are not independently distributed).

  For a given hash function with output size of N bits, this general algorithm succeeds after approximately $\sqrt{pi/2} \times 2N/2$ evaluations of the hash function [1].

  This means that for a hash size of N = 128 bits (as given in the question) finding a collision needs approximately $2^{64} \approx 22.10^{18}$ evaluations, which is currently just in reach for a large computing project but certainly will require a lot of effort on our meagre part. There must be a better way.

- **Cycle Detection Approach:** The basic observation was as follows. let f be **MY60SHA** hash function, M be a randomly chosen plaintext and h the hash of M such that $f(M) = h$. If we continuously use h as M and input it into f .i.e $M = h; f(M) = h'$, at a certain point we can expect to see a cycle emerge. At such a point two different messages M and M' converge into the same hash h .i.e. $f(M') = h' = f(M) = h$

  To illustrate the approach let us take an example, let f be **MY8SHA** hash function (where we just break *Col* in the first 8 bits of the *SHA-1* hash function):

  - Let $M =$ "A" and thus: $f(M) = 6d$
  - Let $M = 6d$ and thus: $f(M) = e4$
  - Let $M = e4$ and thus: $f(M) = 20$
  - Let $M = 20$ and thus: $f(M) = 91$
  - Let $M = 91$ and thus: $f(M) = 4c$
  - Let $M = 4c$ and thus: $f(M) = c4$
  - Let $M = c4$ and thus: $f(M) = e4$
  - Let $M = e4$ and thus: $f(M) = 20$

  Thus we see a cycle emerge at point $M = e4$. This means that $M = 6d, M = c4$ both converge on the value $h = e4$ after going through the function f and we break the *Col* property of our hash function.

  In order to find such a cycle we can take the help of any of the generic cycle detection algorithm.

  Listing 5 shows the python code using Floyd's cycle algorithm [2] and **Listing 6** contains the output of the code.

Listing 5: python code for ex2.3.py

```
import hashlib
import time
import random

SIZE = 15


def getHash(x):
    hash_obj = hashlib.sha1(x.encode())
    result = hash_obj.hexdigest()[: SIZE]
    return result


def floyd(f, x0):
    tortoise = f(x0)  # f(x0) is the element/node next to x0.
    hare = f(f(x0))
    while tortoise != hare:
        tortoise = f(tortoise)
        hare = f(f(hare))
    tortoise = x0
    while tortoise != hare:
        last_x = tortoise
        last_y = hare
        tortoise = f(tortoise)
        hare = f(hare)   # Hare and tortoise move at same speed
    return last_x, last_y, f(last_x)


start = time.time()
print(floyd(getHash, str(random.random())))
end = time.time()
print("Time taken: {}".format(end - start))
```

Listing 6: output of ex2.3.py

```
('46a5ca851c70466', 'fa920e85980b9bb', '80e9c6be101e19c')
Time taken: 65454.24536347389
```

Therefore the strings *"46a5ca851c70466"* and *"fa920e85980b9bb"* both have the same result through hash function **MY60SHA**: *"80e9c6be101e19c"*

Answer - 4:

I have uploaded the picture!

References

[1] van Oorschot, Paul C. and Wiener, Michael J. *Parallel Collision Search with Cryptanalytic Applications*, Journal of Cryptology, 2nd edition, 1999.

[2] Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone *Parallel Collision Search with Cryptanalytic Applications*, Handbook of Applied Cryptography, p. 125