Introduction to Computer Security

Lab 1 - Exploring Symmetric Key Cryptography

Report
of
Andreas Wilhelm

January 29, 2019

You can find the code and everything else also on GitHub under the link *https://github.com/awilsee/CSec*. Maybe more convenient for you.

1 Warm It Up

```
import binascii
from itertools import cycle


def xorfunc(txtCipher, txt):
    if len(txtCipher) != len(txt):
        raise NameError('The two strings hasn\'t equal lenght!')
    return bytes([a ^ b for (a, b) in zip(txtCipher, cycle(txt))])
    #cipher = XOR.new(txtCipher) #only 1 to 32 bytes long
    #return cipher.encrypt(txt)


if __name__ == '__main__':
    txtDarlin = "Darlin dont you go"
    txtHair = "and cut your hair!"


    print("txt:", txtDarlin)
    print("cipher:", txtHair)

    #print("\nencrypted: ", binascii.hexlify(xorfunc(txtHair,
    txtDarlin)))
    print("\nencrypted: ", '[{}]'.format(' ' .join(hex(x) for x in
    xorfunc(bytes(txtHair, 'utf-8'), bytes(txtDarlin, 'utf-8')))))
```

Listing 1: Code of task 1

Firstly implemted the task using pycrypto lib. However this lib has a limit of 32 Byte for the key, so it's not usable for longer strings. Therefore implemented the *xorfunc* by scratch. This functions will be used from the other tasks, as well. Realized the xorfunc with bytewise XOR with a for loop.

2 Implement OTP

```
import binascii
import os

from task1 import xorfunc

filePath = "./txtFile"
filePathEnc = "./txtFileEnc"

file = open(filePath, "rb").read()
print("file {} has {:d} characters" .format(filePath, len(file)))
print("filetxt: ", file)

cipher = os.urandom(len(file))
#print("cipherKeyHEX: ", binascii.hexlify(cipher))

txtEnc = xorfunc(cipher, file)
```

```
17 print("\nencryptedHEX: ", '[{}]'.format(' ' .join(hex(x) for x in
       txtEnc)))
18
19 fileEnc = open(filePathEnc, "wb").write(txtEnc)
20
21 txtDec = xorfunc(cipher, txtEnc)
22
23 print("\ndecrypted: ", txtDec.decode("utf-8"))
```

Listing 2: Code of task 2

For the One Time Pad (OTP) implemented reading and writing files in general. Additionally used the *os.urandom*-function to generate the cipher key with the same length as the file. For encrypting and decrypting the key with the file the *xorfunc* of listing 1 is used.

```
1 Input: "This is a textfile with some input over 32 Bytes.."
2 Hexadecimal output: "[0x6 0x6c 0x83 0x82 0x80 0x3e 0x8e 0x74 0x3a 0
      x8 0x50 0x98 0xe9 0xc4 0x65 0xa5 0xe 0xa2 0xba 0xcb 0xc1 0xa5 0
      xa7 0x86 0xa 0x1c 0xbb 0x21 0x94 0x1b 0x81 0x68 0x8c 0xad 0xf6 0
      xff 0x1e 0xdf 0x38 0xc8 0xa6 0x2b 0x1d 0x16 0x90 0xc1 0x5d 0x92
      0xe5 0x43 0x3b]"
```

3 Encrypt an Image Using OTP

```
1 import binascii
2 import os
3
4 from task1 import xorfunc
5
6 pic1 = "./cp-logo.bmp"
7 pic1Enc = "./cp-logo-enc.bmp"
8 pic2 = "./mustang.bmp"
9 pic2Enc = "./mustang-enc.bmp"
10 pic3Enc = "./random-enc.bmp"
11 picDec = "./pic-dec.bmp"
12
13 fileLength = len(open(pic1, "rb").read())
14
15
16 def write_bmp_file(filename, input):
17     byteEnc = open("./cp-logo.bmp", "rb").read(54)
18     byteEnc += input[54:]
19     open(filename, "wb").write(byteEnc)
20
21
22 def encryptBMPFile(picFilePath, picEncFilePath, cipher=os.urandom(
       fileLength)):
23     file = open(picFilePath, "rb").read()
24     print("file {} has {:d} characters".format(picFilePath, len(file
       )))
25
26     txtEnc = xorfunc(cipher, file)
27     # print("\nencryptedHEX: ", '[{}]'.format(' ' .join(hex(x) for x
        in txtEnc)))
28
29     write_bmp_file(picEncFilePath, txtEnc)
30
```

```
31      txtDec = xorfunc(cipher, txtEnc)
32
33      open(picDec, "wb").write(txtDec)
34
35
36  def generateRndBMPFile(picEncFilePath, cipher=os.urandom(fileLength)
        ):
37      write_bmp_file(picEncFilePath, cipher)
38
39
40  if __name__ == '__main__':
41      encryptBMPFile(pic1, pic1Enc)
42      encryptBMPFile(pic2, pic2Enc)
43      generateRndBMPFile(pic3Enc)
```

Listing 3: Code of task 3

The function *write_bmp_file* is used for writing the encrypted files with the original BMP-Header, so that it's possible to read it with an image viewer.
For encrytion of the BMP-File the function *encryptBMPFile* was implemented which uses again the *xorfunc* of listing 1 and the previous mentioned one for saving.
With *generateRndBMPFile* a BMP-File with random input will be created.

Figure 1: encrypted pictures

(a) CP-Logo (b) Mustang (c) Random

What do you observe?
You get a lot of random data where you can't see anything.

Are you able to derive any useful information about from either of the encrypted images? What are the causes for what you observe?
You can't derive any useful information out of this images, because you XOR the data with different random keys which results in random data again.

4 The Two-Time Pad

```
1  import os
2
3  import task3
4
```

```
5
6  def xorImages(pic1FilePath, pic2FilePath):
7      file1 = open(pic1FilePath, "rb").read()
8      file2 = open(pic2FilePath, "rb").read()
9      print("file {} has {:d} characters".format(pic1FilePath, len(
       file1)))
10
11     txtEnc = task3.xorfunc(file1, file2)
12     # print("\nencryptedHEX: ", '[{}]'.format(' ' .join(hex(x) for x
       in txtEnc)))
13
14     task3.write_bmp_file("./xorPics.bmp", txtEnc)
15
16
17 if __name__ == '__main__':
18     cipher = os.urandom(task3.fileLength)
19     task3.encryptBMPFile(task3.pic1, task3.pic1Enc, cipher)
20     task3.encryptBMPFile(task3.pic2, task3.pic2Enc, cipher)
21     task3.generateRndBMPFile(task3.pic3Enc, cipher)
22     xorImages(task3.pic1Enc, task3.pic2Enc)
```

Listing 4: Code of task 4

EncryptBMPFile of listing 3 is used again for encryption of the BMP-Files with the same cipher key. Afterwards with function *xorImages* both encrypted images are XOR together.

Figure 2: encrypted pictures

(a) CP-Logo (b) Mustang (c) XOR a und b

Are you able to now derive any useful information about the original plaintexts from the resulting image? What are the causes for what you observe?
Now you can see clearly recognize both pictures in figure 3c. The resulting picture is basically the first and second originally picture XOR together. Because if you XOR two times the same and with another XOR you get the same result as you only XOR the last one. And exactly that happend because we encrypted both pictures with the same key.

5 Modes of Operation

```
1 import os
2 from Crypto.Cipher import AES
3
4 from task3 import write_bmp_file
```

```
5  from task1 import xorfunc
6
7  txtfilePath = "./txtFile"
8  pic1 = "./cp-logo.bmp"
9  pic2 = "./mustang.bmp"
10
11 ending_ecb = "-enc-ecb.bmp"
12 ending_cbc = "-enc-cbc.bmp"
13
14 aes_bytes = 16
15 cipher_key = os.urandom(aes_bytes)
16 aes_prim = AES.new(cipher_key)
17
18
19 def add_pkcs7(input):
20     num_bytes = aes_bytes - (len(input) % aes_bytes)
21     fil_bytes = bytes()
22     for x in range(num_bytes):
23         fil_bytes += bytes([num_bytes])
24     input += fil_bytes
25     return input
26
27
28 def get_padded_file(file_path):
29     file = open(file_path, "rb").read()
30     print("file {} has {:d} characters".format(file_path, len(file))
       )
31
32     # padding
33     return add_pkcs7(file)
34
35
36 def encrypt_ecb(data):
37     x = 0
38     file_enc = bytes()
39     while x < len(data) / aes_bytes:
40         file_enc += aes_prim.encrypt(data[x * aes_bytes:((x + 1) *
       aes_bytes)])
41         x += 1
42     return file_enc
43
44
45 def encrypt_ecb_file(file_path):
46     return encrypt_ecb(get_padded_file(file_path))
47
48
49 def decrypt_ecb(data):
50     x = 0
51     file_enc = bytes()
52     while x < len(data) / aes_bytes:
53         file_enc += aes_prim.decrypt(data[x * aes_bytes:((x + 1) *
       aes_bytes)])
54         x += 1
55     pad_byte = file_enc[len(file_enc) - 1]
56     return file_enc[:len(file_enc) - pad_byte]
57
58
59 def encrypt_cbc(data, init_iv):
```

```
60      x = 0
61      file_enc = bytes()
62      while x < len(data) / aes_bytes:
63          if 0 == x:
64              xor_bytes = xorfunc(init_iv, data[x * aes_bytes:((x + 1)
        * aes_bytes)])
65          else:
66              xor_bytes = xorfunc(file_enc[(x - 1) * aes_bytes:(x *
      aes_bytes)], data[x * aes_bytes:((x + 1) * aes_bytes)])
67          file_enc += aes_prim.encrypt(xor_bytes)
68          x += 1
69      return file_enc
70
71
72  def encrypt_cbc_file(file_path, init_iv):
73      return encrypt_cbc(get_padded_file(file_path), init_iv)
74
75
76  def decrypt_cbc(data, init_iv):
77      x = 0
78      dec_data = bytes()
79      while x < len(data) / aes_bytes:
80          dec_bytes = aes_prim.decrypt(data[x * aes_bytes:((x + 1) *
      aes_bytes)])
81          if 0 == x:
82              dec_data += xorfunc(init_iv, dec_bytes)
83          else:
84              dec_data += xorfunc(dec_bytes, data[(x - 1) * aes_bytes
      :(x * aes_bytes)])
85          x += 1
86      pad_byte = dec_data[len(dec_data) - 1]
87      return dec_data[:len(dec_data) - pad_byte]
88
89
90  if __name__ == '__main__':
91      enc_txt = encrypt_ecb_file(txtfilePath)
92      open(txtfilePath + ending_ecb[:8], "wb").write(enc_txt)
93
94      #test decyrpt ecb
95      cipher = AES.new(cipher_key, AES.MODE_ECB)
96      plaintext = cipher.decrypt(enc_txt)
97      print(plaintext)
98
99      #test own decrypt implementation
100     print(decrypt_ecb(enc_txt))
101
102     enc_bytes = encrypt_ecb_file(pic1)
103     write_bmp_file(pic1 + ending_ecb, enc_bytes)
104
105     enc_bytes = encrypt_ecb_file(pic2)
106     write_bmp_file(pic2 + ending_ecb, enc_bytes)
107
108     init_iv = os.urandom(aes_bytes)
109     enc_txt = encrypt_cbc_file(txtfilePath, init_iv)
110     open(txtfilePath + ending_cbc[:8], "wb").write(enc_txt)
111
112     #test decyrpt cbc
113     cipher = AES.new(cipher_key, AES.MODE_CBC, init_iv)
```

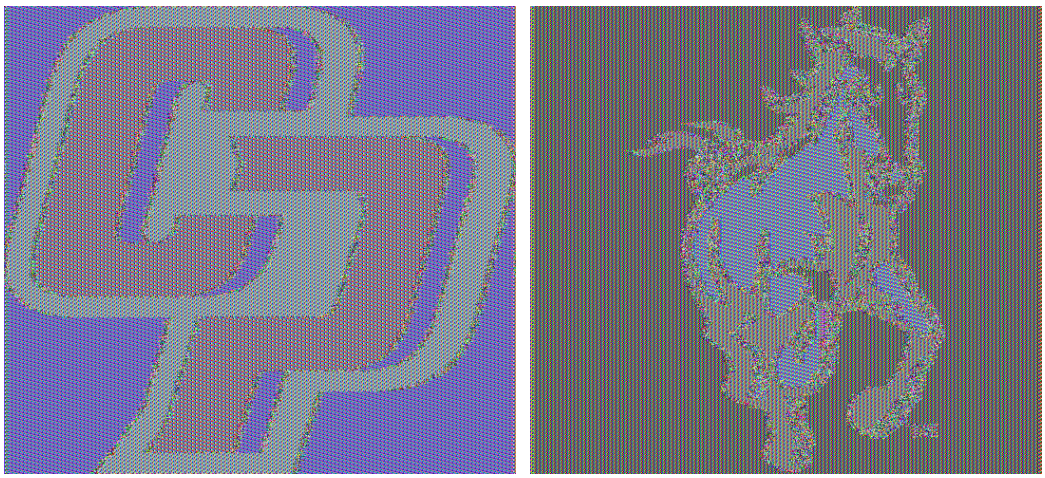
```
114     plaintext = cipher.decrypt(enc_txt)
115     print(plaintext)
116
117     #test own decrypt implementation
118     print(decrypt_cbc(enc_txt, init_iv))
119
120     enc_bytes = encrypt_cbc_file(pic1, init_iv)
121     write_bmp_file(pic1 + ending_cbc, enc_bytes)
122
123     enc_bytes = encrypt_cbc_file(pic2, init_iv)
124     write_bmp_file(pic2 + ending_cbc, enc_bytes)
```

Listing 5: Code of task 5

The first two functions are for adding the PKCS#7 padding, so to fill up the 128-bits for the encryption.
In the *encrypt_ecb* function is the operation mode of ECB implemented where each separate block of data will be encrypted by using the encrypt function of the cryptolib. The same cipher key is used for the whole program.
The decryption is basically the same as encryption in ECB-Mode except additionally the padding files will be removed.

In operation mode of CDC in each block of data the plaintext will be firstly XOR together either with INIT_IV at the beginning or with the encrypted ciphertext block. After the XOR the encryption function of cryptolib is called.
The decryption is similiar. At first the block will be decrypted. Afterwards XOR together with INIT_IV at the beginning or the previous ciphertext block.
To see that the own implementation of the encryption and decryption algorithm works correctly it was prooved with the cryptolib-function.

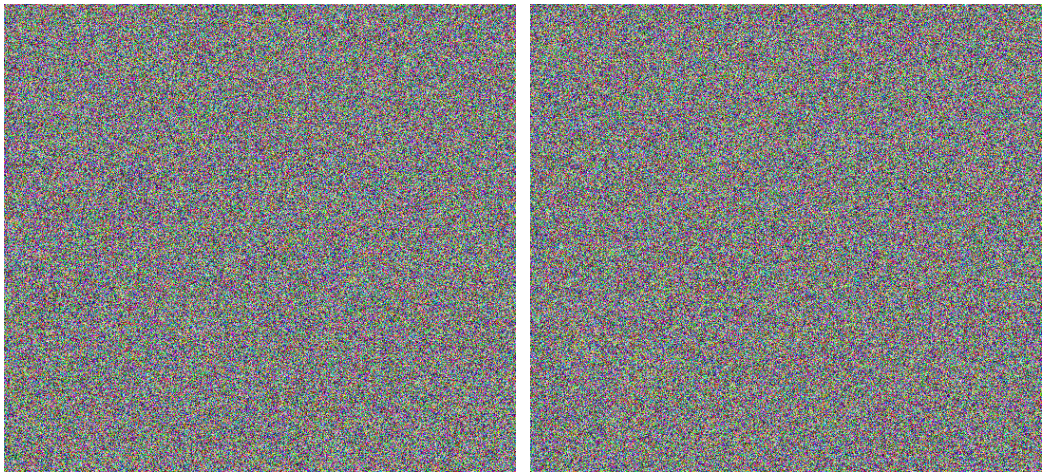
Figure 3: encrypted pictures in ECB



(a) CP-Logo (b) Mustang

What do you observe? Are you able to derive any useful information about from either of the encrypted images? What are the causes for what you observe?
With ECB mode you can see in figure 4a and 4b that you clearly can recognize what's in this picture. You don't have the image quality and colors but you definitely can see how it looks like. Well, the whole picture is encrypted with the exact same key and prozedure.

Figure 4: encrypted pictures in CBC



(a) CP-Logo (b) Mustang

So there will be some bits flipped which results in the change of color.
In CBC Mode as shown in figure 5a and 5b it's random data, because you always take different data for the XOR function.

6 The Limits of Confidentiality

```
1  import os
2
3  from task5 import add_pkcs7
4  from task5 import encrypt_cbc
5  from task5 import decrypt_cbc
6
7  aes_bytes = 16
8  init_iv = os.urandom(aes_bytes)
9
10
11 def submit(data):
12     url = "userid=321;userdata=" + data + ";session-id=31337"
13     for i in range(len(url)):
14         if url[i] == ";" or url[i] == "=":
15             url = url.replace(url[i], "*")
16     #url = urllib.parse.quote(url, safe="\' ',")
17
18     url_bytes = add_pkcs7(bytes(url, 'utf-8'))
19     return encrypt_cbc(url_bytes, init_iv)
20
21
22 def verify(data):
23     url = decrypt_cbc(data, init_iv)
24     #print(url)
25     if -1 == url.decode('utf-8', 'ignore').find(";admin=true;"):
26         return False
27     else:
28         return True
29
```

```
30
31 # modify cipher block to modify following plaintext
32 def modify_block(enc_data):
33     cipher_list = []
34     i = 0
35     # get blocks
36     while i * 16 < len(enc_url):
37         cipher_list.append(enc_url[i * 16: 16 + (i * 16)])
38         i += 1
39
40     # modify cipher block before actual plaintext block
41     block_to_modify = cipher_list[0]
42     block_list = list(block_to_modify)
43     block_list[4] ^= ord("*") ^ ord(";")
44     block_list[10] ^= ord("*") ^ ord("=")
45     block_list[15] ^= ord("*") ^ ord(";")
46     cipher_list[0] = bytes(block_list)
47     return b''.join(cipher_list)
48
49
50 if __name__ == '__main__':
51     #txt = "You're the man now, dog"
52     txt = ";admin=true"
53     #txt = input("Enter string: ")
54     print("Entered String: " + txt)
55
56     #submit and encrypt input string
57     enc_url = submit(txt)
58
59     #modify block
60     enc_url = modify_block(enc_url)
61
62     #call verify
63     if verify(enc_url):
64         print("\nCongrats! You're admin!")
65     else:
66         print("\nYou're NOT admin with your string!")
```

Listing 6: Code of task 6

Firstly in the *submit*-function the url is created with the payload. Afterwards the unwanted characters will be removed. For padding and encryption of this url text the functions of listing 5 is used.
In the *verify*-function the data will be decrypted with previous implemented function of listing 5 and checked for the substring.
To modify the encrypted data in *modify_block*, the data will be firstly truncated into blocks of 128-bit. Now the ciphertext block before the actual plaintext block, where you wanna modify some bits, will be some bits flipped. You have to XOR the byte with the current plaintext sign and XOR it again with the sign you want because if you XOR with the same sign and afterwards with your own, only your own sing will be left after decryption.

Why was this attack possible? What would this scheme need in order to prevent such an attack?
This attack is possible because it's always only dependent of the block before and the XOR function and there is no integrity of the data which is exactly used in this exploit.
A MAC or HMAC or an other authenticated encryption algorithm can be used to prevent

this attack.

7 Exploring Cryptographic Hash Functions

7.1 One-wayness

```
import hashlib
import math
import os
import time
import matplotlib.pyplot as plt

def one_wayness():
    #txt = input("Enter string to hash: ")
    #print(hashlib.sha256(txt.encode()).hexdigest())

    print("10 example hashes:")
    for i in range(10):
        print(hashlib.sha256(os.urandom(256)).hexdigest())


    print("\n2 hashes with hamming distance:")
    print(hashlib.sha256("Example string 1".encode()).hexdigest())
    print(hashlib.sha256("Example string 2".encode()).hexdigest())
```

Listing 7: Code of task 7

What do you observe?
The hash-function always creates completely different hexnumbers of 256-bits length. Despite you only change one bit in the string (Hamming Distance == 1) the hash is completely different. *How many of the bytes are different between the two digests?*
Mostly all of them.

7.2 Preimage Resistance

```
def preimage_res(target):
    print("Call for target: 0x{:064X}" .format(target))

    i = 1
    while True:
        hash_val = int.from_bytes(hashlib.sha256(os.urandom(20)).digest(), 'big')
        if target > hash_val:
            break
        i += 1
    print("#Inputs: {}\n targetVal: 0x{:064X}\n digest: \t0x{:064X}\n" .format(i, target, hash_val))
    return i
```

Listing 8: Code of task 7

After the conversion of the byte-output to an integer the hash can be compared against the target numbers.

What do you observe about the number of inputs required?
As shown in table 1 the number of required imputs increase exponentially, because more

Target	0x0X	0x00X	0x000X	0x0000X	0x00000X
#Inputs	20	196	984	83765	114546

Table 1: results

bytes have to be the same, where the probability is decreasing.

7.3 Collision Resistance

```
def collision_res():
    num_of_loops = 5

    bits_list = [list() for x in range(num_of_loops)]
    dict_list = [list() for x in range(num_of_loops)]
    time_list = [list() for x in range(num_of_loops)]

    dict_hashs = dict()

    for i in range(num_of_loops):
        print("test {}".format(i))
        bits = 8
        while 50 >= bits:
            bits_list[i].append(bits)
            start_time = time.process_time()
            while True:
                value = os.urandom(20)
                # generate key and get only desired number of bytes
                key = int.from_bytes(hashlib.sha256(os.urandom(20)).
    digest()[0:(math.ceil(bits / 8))], 'big')
                # shift bytes so only 12 bits are left for
    comparison
                key = key >> ((math.ceil(bits / 8) * 8) - bits)
                # create dict
                if key in dict_hashs:
                    if value != dict_hashs[key]:
                        elapsed_time = (time.process_time() -
    start_time)
                        time_list[i].append(elapsed_time)
                        dict_list[i].append(len(dict_hashs) / 1000)
                        print("Found duplicate with {} bits, needed
    {:.0f} s #inputs {}k\nhash1: 0x{:020X}\nInput1: 0x{:040X}\
    nInput2: 0x{:040X}\n" .format(bits, elapsed_time, len(dict_hashs
    ) / 1000, key, int.from_bytes(dict_hashs[key], 'big'), int.
    from_bytes(value, 'big')))
                        break
                else:
                    dict_hashs[key] = value
            dict_hashs.clear()
            bits += 2

    #calculate mean
    bits_mean_list = []
    dict_mean_list = []
    time_mean_list = []
    for i in range(len(bits_list[0])):
        bits_mean = 0
```

```
41          dict_mean = 0
42          time_mean = 0
43          for j in range(num_of_loops):
44              bits_mean += bits_list[j][i]
45              dict_mean += dict_list[j][i]
46              time_mean += time_list[j][i]
47          bits_mean_list.append(bits_mean/float(num_of_loops))
48          dict_mean_list.append(dict_mean/float(num_of_loops))
49          time_mean_list.append(time_mean/float(num_of_loops))
50
51      print(bits_mean_list)
52      print(dict_mean_list)
53      print(time_mean_list)
54
55      #plotting graphs
56      fig = plt.figure()
57      for j in range(num_of_loops):
58          plt.plot(bits_list[j], dict_list[j], ':')
59      plt.plot(bits_mean_list, dict_mean_list, color='red')
60      plt.xlabel('digest size [bits]')
61      plt.ylabel('# of inputs [k]')
62      fig.savefig('plot_num_input.png', dpi=500, bbox_inches='tight')
63      fig.savefig('plot_num_input.svg', format='svg', dpi=500,
    bbox_inches='tight')
64
65      fig1 = plt.figure()
66      for j in range(num_of_loops):
67          plt.plot(bits_list[j], time_list[j], ':')
68      plt.plot(bits_mean_list, time_mean_list, color='red')
69      plt.xlabel('digest size [bits]')
70      plt.ylabel('collision time [s]')
71      fig1.savefig('plot_time.png', dpi=500, bbox_inches='tight')
72      fig1.savefig('plot_time.svg', format='svg', dpi=500, bbox_inches
    ='tight')
```

Listing 9: Code of task 7

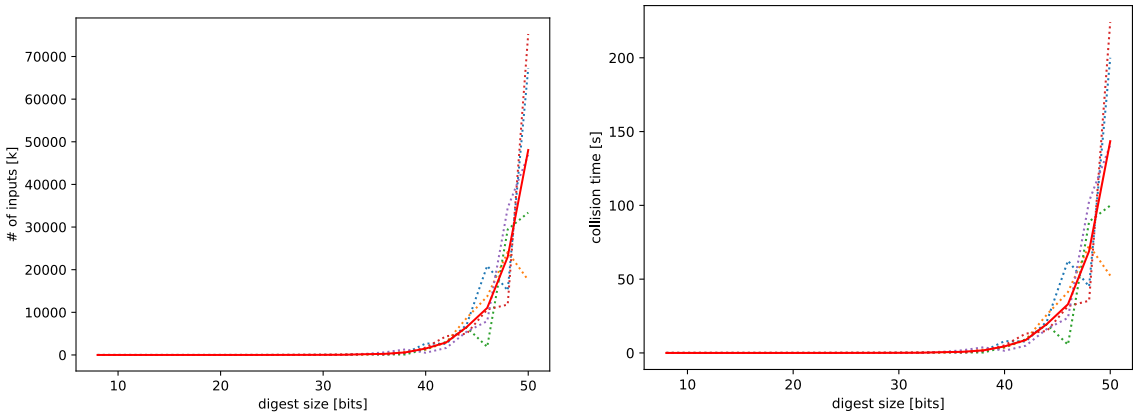
The task was to get the same sequence (several runs with different length of same sequence) of a hash twice with different input. Therefore new hashes will be generated and stored in a dictionary as long as two sequences of hashes matches together. To get the elapsed time a timestamp in line 15 in listing 9 directly before the generation fo the hashes will be taken and after a hash match with another one and where the two inputs were different. The time function consider only the processing time, so not the sleeping time. This measurements was running on a Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz.

For less memory usage and better performance only the sequence of the hash will be stored and the input number is limited to 20 Bytes.
For better statistics the whole prozedure was run 5 times with all different length from 8 bits till 50 bits in 2 bit steps and the average was calculated. And even with that I got to the limit of my computer with 16GB memory space and 52-bit sequence.

On figure 6a and 6b you can see that both are equal which also makes sense because the time is absolutely related to the number of inputs, as only the processing time was measured. So if you need more inputs you need more time because the computer can handle x inputs in a second, there is no variance.

In the graphs are the pointed lines the exact measurements and the red drawn through line is the average between them.
You can see that the time consumption respectively the needed number of input rises exponentially with the digest size.

Figure 5: Plots of the #inputs and time corresponding to digest size



(a) digest with #inputs

(b) digest with time

What is the maximum number of files you would ever need to hash to find a collision on an n-bit digest?
By the pigeon hole principle, we know that if we hash N+1 items, two of them are certain to have the same hash value.

Given the Birthday Bound, what is the expected number of hashes before a collision on an n-bit digest?
With the Birthday Bound it would be $\sqrt{n}$.

Is this what you observed? Given the data you've collected, speculate on how long it might take to find a collision on the full 256-bit digest.
Sure! It's obvious that a copmuter needs way longer the longer the same digest size is. The probability to find an equal hash decreases significantly.

Given an 8-bit digest, would you be able to break the one-way property (i.e. can you find any pre-image)?
With the 256-bit hash function you will take forever. And even if you find a number which results in the same hash, it could be the wrong one because of the collisions.

Do you think this would be easier or harder (i.e. more or less work) than finding a collision? Why or why not?
It's way more work and need way more time to calculate all posibilities. With the collisions and using the Birthday Bound you can look for any collision, doesn't matter with which hash. With the one-way you have to find exactly this one.