Introduction

Birthday attack is a type of cryptographic attack that belongs to a class of brute force attacks. It exploits the mathematics behind the birthday problem in probability theory. The success of this attack largely depends upon the higher likelihood of collisions found between random attack attempts and a fixed degree of permutations, as described in the birthday paradox problem.

The birthday attack is a generic algorithm that is used to create hash collisions. Just as matching your birthday is difficult, finding a specific input with a hash that collides with another input is difficult. However, just like matching any birthday is easier, finding any input that creates a colliding hash with any other input is easier due to the birthday attack.

Problem Statement

In this project, we implement a program that runs a birthday attack against a hash function called "BadHash40". BadHash40 is constructed using SHA256 as a subroutine by truncating the output of the SHA256 function that outputs the first 40 bits of its argument. The construction of BadHash40 is illustrated in the Figure 1 below.

Our goal is to find two (arbitrary) inputs to the function BadHash40 which result in the same output, in other words, to find a collision for BadHash40 and then we output these inputs and the corresponding BadHash40 output in the hex format into a text file named hash.data.

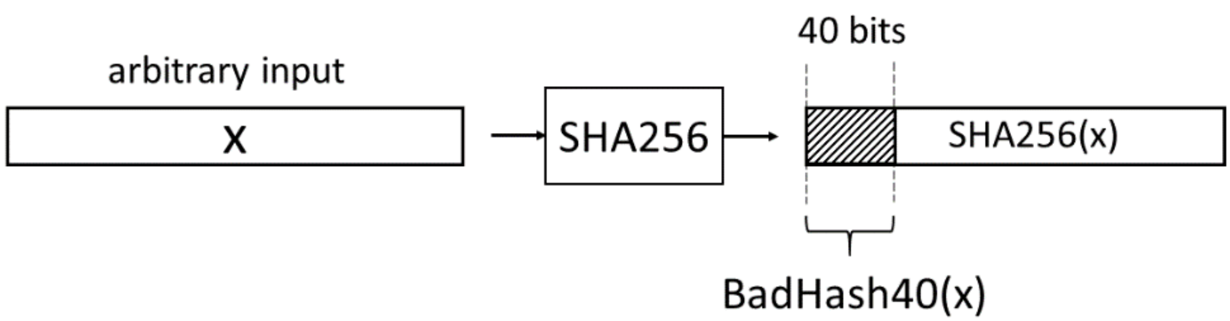


Figure 1: The construction of BadHash40

Methodology

The code is written in Python 3.7.x and uses the SHA256 function from the hashlib library to search for collisions. It takes one argument: 'lhash' which is parsed into the program from the command line using the flag option '-lhash' and it sets the length of the hash input/output list.

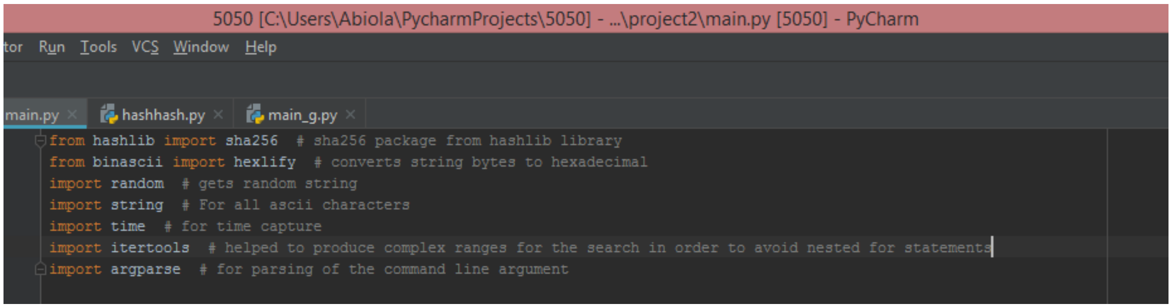
The way the code works is this; random hashes are generated, and the results of each hash are stored as keys in a dictionary (Python's implementation of the hash table data structure). This allows lookup of collisions for already generated hashes to happen in constant time, saving valuable time compared to initially saving to file and then reading out to compare. When a collision is found, the two colliding inputs as well as their conflicting hashes are printed to screen in addition to the total number of evaluations it took to find the hash and the time taken in seconds.

In addition to the display on the console, the entire input/output pairs as well as the colliding inputs and their hashes are also stored in a text file. These results are stored in hexadecimal format.

Description of Program Code

Here, we show a brief description of various code components of that that made up the programs.

The Libraries Imported

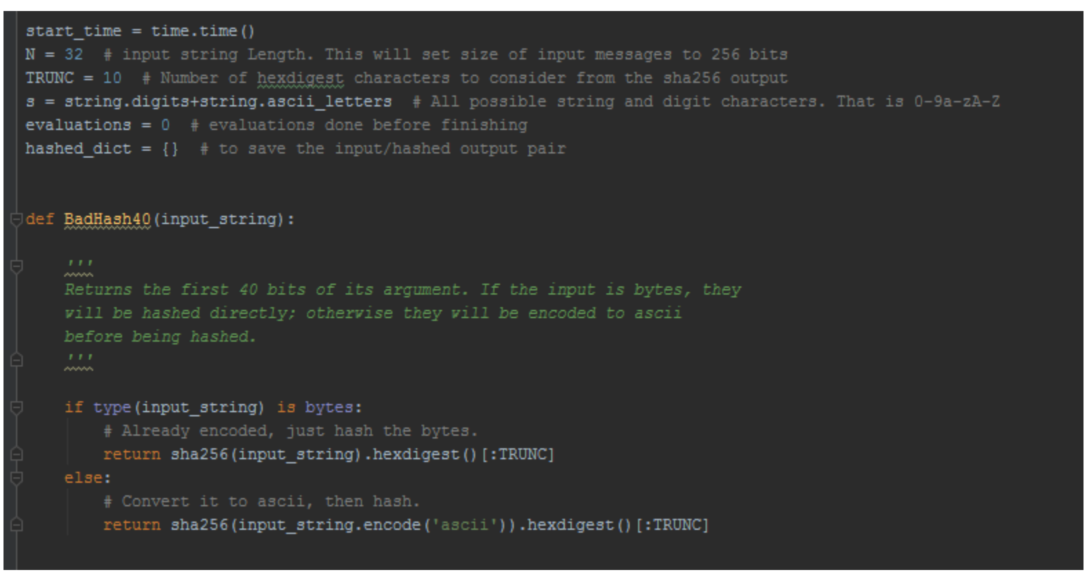



```python
from hashlib import sha256  # sha256 package from hashlib library
from binascii import hexlify  # converts string bytes to hexadecimal
import random  # gets random string
import string  # For all ascii characters
import time  # for time capture
import itertools  # helped to produce complex ranges for the search in order to avoid nested for statements
import argparse  # for parsing of the command line argument
```

Figure 2. The library packages used for the program

Figure 2 above shows all the necessary python libraries that were applied. *Hashlib* library contains the SHA256 function which we used to create the hashes of the inputs. The hexlify method of the *binascii* library was used to convert the output results into hexadecimal formats, *random* library was used to generate random input messages of desired amount. The string library was used to import all the ascii characters including the lower and upper case alphabets and numerical digits. *Time* package was employed to calculate the tame taken to find a collision, the *itertools* package was used t produce complex ranges for the collision search in order to avoid nested *for statements* there optimizing the program and lastly, the *argparse* library was needed to parsing the command line argument 'lhash' into program.

The badHash40 Function



```python
start_time = time.time()
N = 32  # input string Length. This will set size of input messages to 256 bits
TRUNC = 10  # Number of hexdigest characters to consider from the sha256 output
s = string.digits+string.ascii_letters  # All possible string and digit characters. That is 0-9a-zA-Z
evaluations = 0  # evaluations done before finishing
hashed_dict = {}  # to save the input/hashed output pair


def BadHash40(input_string):

    '''
    Returns the first 40 bits of its argument. If the input is bytes, they
    will be hashed directly; otherwise they will be encoded to ascii
    before being hashed.
    '''

    if type(input_string) is bytes:
        # Already encoded, just hash the bytes.
        return sha256(input_string).hexdigest()[:TRUNC]
    else:
        # Convert it to ascii, then hash.
        return sha256(input_string.encode('ascii')).hexdigest()[:TRUNC]
```

Figure 3. The BadHash40 Function Code

The BadHash40 function accepts a 32 byte message as input and returns a 40 bit hashed output which is a result of the truncation of the SHA256 output of the string. If the input message is already in bytes, it is hashed directly, otherwise, it is first encoded into bytes before hashing.

The get_random function

```python
def get_random():
    '''
    Get random characters from s of length N and returns rand_input of 32 characters
    '''
    rand_input = ''.join(random.sample(s, N))
    return rand_input
```

Figure 4: The get_random function

The get_random function shown in Figure 4, randomly generates 256 bits messages (32 characters) from a permutation of the ascii characters which serves as input to the BadHash40 function.

The Collision Finder

```python
'''
This block of code finds the collision.
NN represents the -lhash parameter which sets the size of input/output pairs.
It generates input messages of size NN, hash them and stores in a dictionary.
It then checks for collision among the hashes henerated, if collision found, it
outputs the inputs and their collision, else, it start afresh by generating new list
of input/output pairs and store in the dictionary.
'''
parser = argparse.ArgumentParser()
parser.add_argument("-lhash", required=True)
NN = int(parser.parse_args().lhash)
ranges = [range(NN), range(NN)]

for i, j in itertools.product(*ranges):
    evaluations += 1
    message = get_random()  # gets a random message input
    message2hex = str(hexlify(message.encode("ascii")))  # converts to 64 hexadecimal characters
    message_hash = BadHash40(message)

    if message_hash not in hashed_dict.keys():
        try:  # Let's expect a MemoryError if we search a too big collision
            hashed_dict[message_hash] = message
        except MemoryError:
            print("LOG: MemoryError")
            hashed_dict = {}
    else:
        if hashed_dict[message_hash] == message:
            print("String Already Used!")
            print("Number of evaluations made so far", evaluations)
        else:
            break
```

Figure 4: The block of Code that Finds the Collision

Figure 4 above shows the block of code that finds the collision. A dictionary is populated with the input and corresponding output hash pairs of size equal to the NN parameter. NN represents the -lhash parameter which sets the size of input/output pairs given from the command line.

It then checks for collision among the hashes generated, if collision found, it breaks the loop and output the inputs and their collision, otherwise, the process is repeated generating a new list of input/output pairs to be stored in the dictionary and then it searches for collision again.

The Result Display

```
'''
If a collision is found, the colliding inputs are identified using the dictionary keys method
and they are converted to hexadecimal format before display on console or saved to file.

Opens the hash.data file and creates the header for the inputs and their hashes
stores the collided messages in hex format and also populates the entire dictionary which holds the
input/output pairs
'''
colliding = hashed_dict[message_hash]
colliding2hex = str(hexlify(colliding.encode("ascii")), "ascii")  # converts to 64 hexadecimal characters
collHash = BadHash40(colliding)
os.system('clear')

with open('hash.data', 'a+') as f:
    f.writelines('----------------------------------------------------------------'
                 '-Collision Found!----------------------------------------------\n')
    f.writelines(['Trial Message(in Hex): ', colliding2hex, '\tTrial Message Hash: ', collHash,
                  '\nOriginal Message(in Hex): ', message2hex, '\tOriginal Message Hash: ', message_hash, '\n'])
    f.writelines('-' * 126)
    f.writelines(['\n', 'Trial Message'.ljust(64, ' '), '\tTrial Message Hash'.ljust(10, ' '), '\n'])
    for key in hashed_dict:
        hashed_dict[key] = str(hexlify(hashed_dict[key].encode("ascii")))
        in_out_pairs = [hashed_dict[key], '\t\t', key, '\n']
        f.writelines(in_out_pairs)
print('--------------------------Collision Results---------------------------')
print("                         Number of evaluations made: ", str(evaluations)+'  ')
print("                            Time Taken: %.2f seconds   " % (time.time() - start_time))
print('                                   Input(s)'.ljust(64, ' ') + ' | '+'Hash'.ljust(10, ' '))
print('-----------------------------------------------------------------+-------')
print(colliding2hex+' | '+collHash+'\n' +message2hex+' | '+message_hash)
print('-----------------------------------------------------------------+-------')
```

Figure 6: The block of Code that Displays/Stores Result

The collision result is outputted in two ways, the first is the display on console showing the two input messages that resulted in collision, their hashes, the time in seconds it took to find the collision and the number of evaluations attempted before a collision was found. While the second is the storing of the results into the hash.data text file. The entire input/output pairs in the dictionary is stored and also the two messages that collided are identified together with their hashes.

<u>Results</u>

Figures 7a-e below shows the results from the collision program. On the average, it takes about 40 seconds to find run and output the collision result which can be observed from the time reported on the images of Figure 7a, b and c. Also, averagely, it requires about a million evaluations to find the collision. On few occasions, it may take longer in cases where it had to take more take one-round and this also increases the time taken.

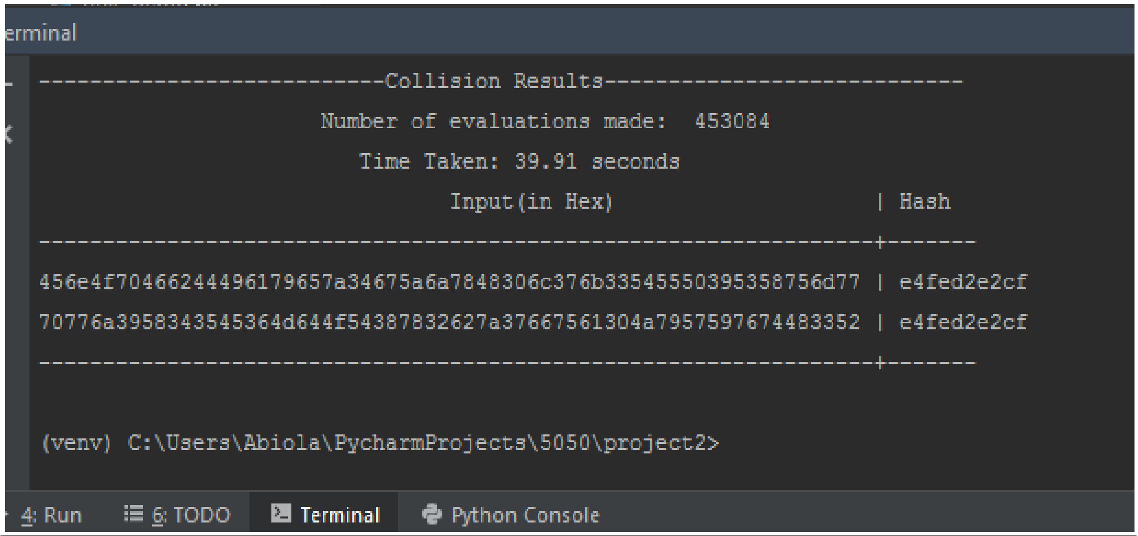


Figure 7a: Display Output Showing Collisions from Two Inputs (Pycharm IDE) - First Run

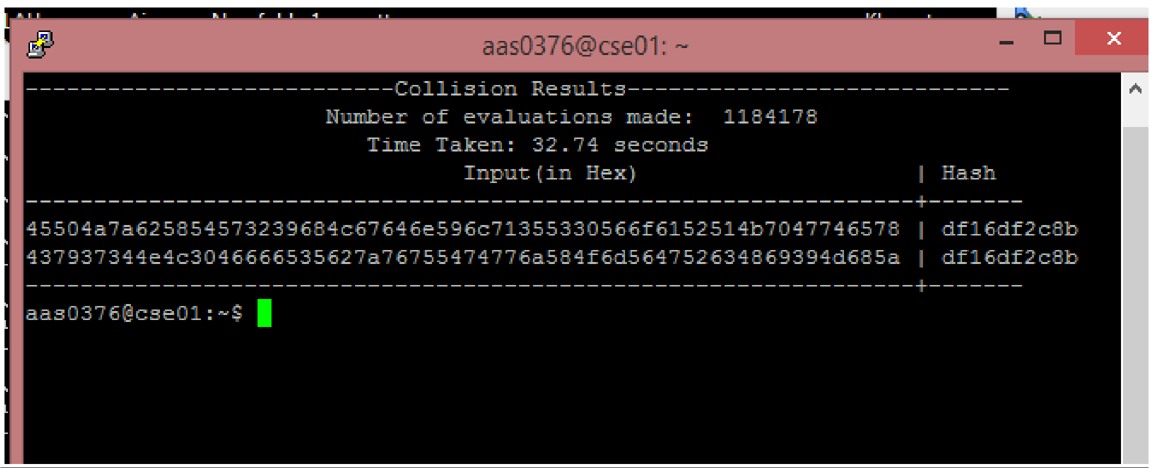


Figure 7b: Display Output Showing Collisions from Two Inputs (CSE Machine) - Second Run

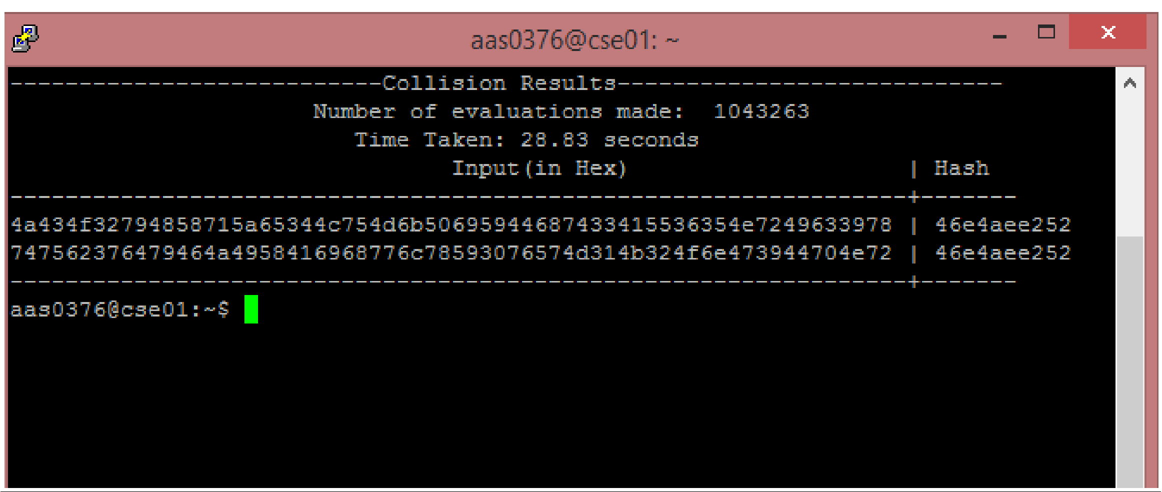


Figure 7c: Display Output Showing Collisions from Two Inputs (CSE Machine) - Third Run

Figure 7d shows the hash.data file where the input and hashed output pairs are stored. Stored also in this file is the two messages that collided along with their conflicting hashes.

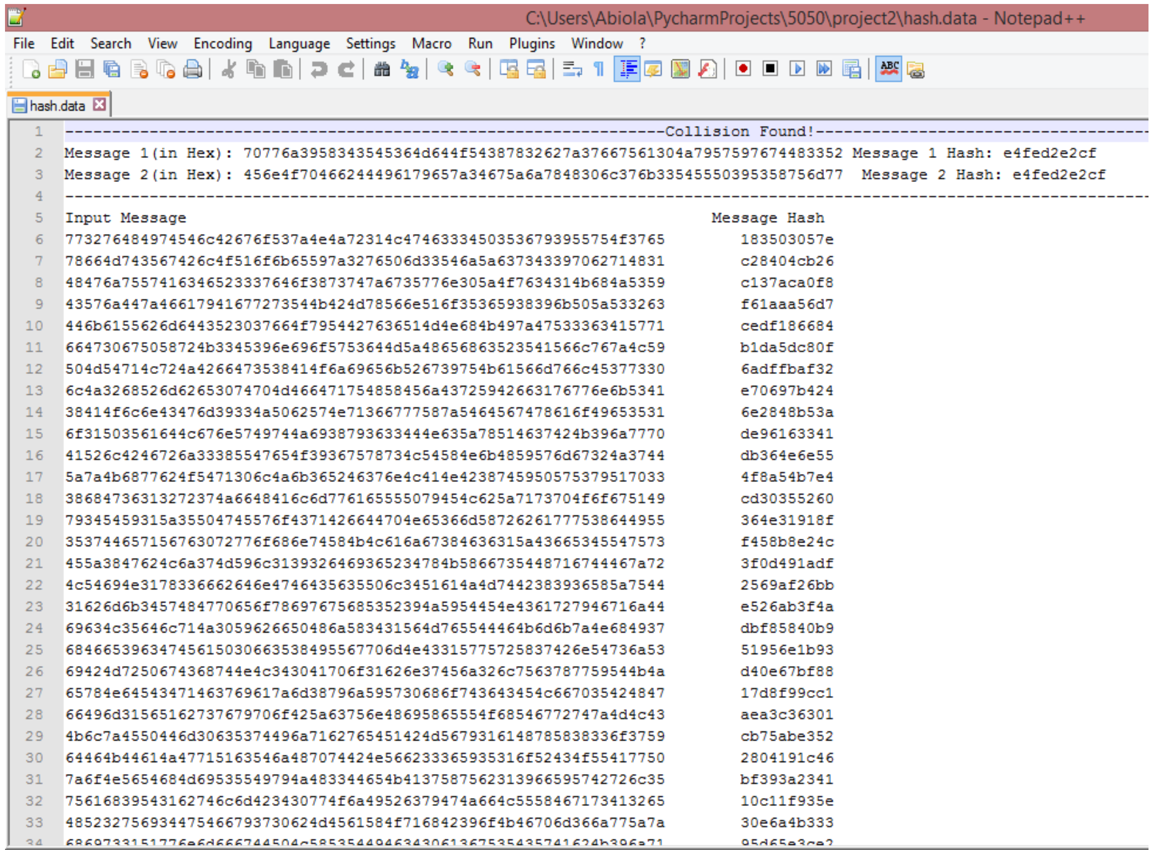


Figure 7d: Display Output Showing the Collided Inputs and the Input/Output Pairs

Figure 7e below shows the usage error. If the flag parameter or the '-lhash' key word is omitted, the user is alerted on the requirement for running the program.

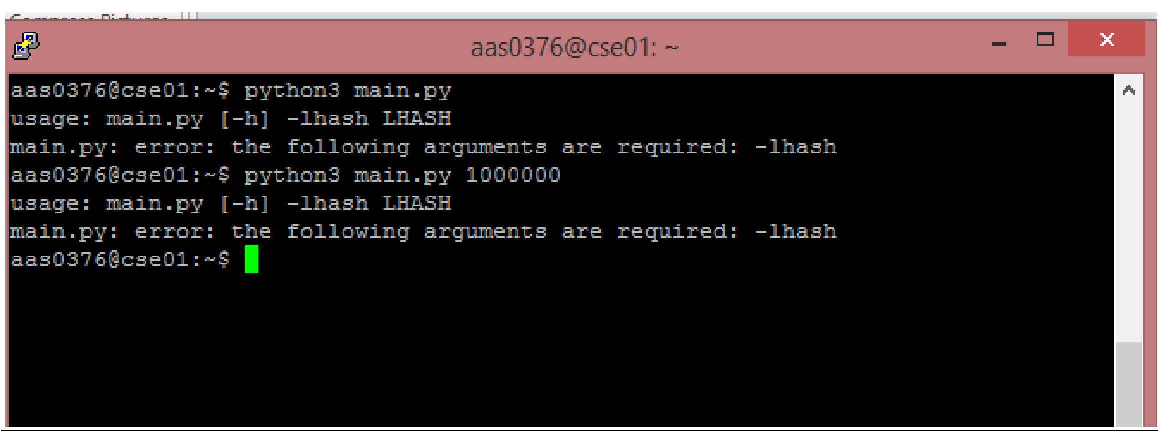


Figure 7e: Display Output Showing the usage Message

Conclusion

This birthday attack program code justifies that for a badly designed or constructed hash function, two inputs that will hash to the same output are easy to find. This reiterates the point that you should not be designing your own cryptographic functions without proper knowledge.

References

https://docs.python.org/2/library/argparse.html

https://www.geeksforgeeks.org/birthday-attack-in-cryptography

https://github.com/nathantypanski/sha1-collisions/blob/master

https://github.com/MarcoMorella/SHA1Collider/blob/master

https://realpython.com/iterate-through-dictionary-python