

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.

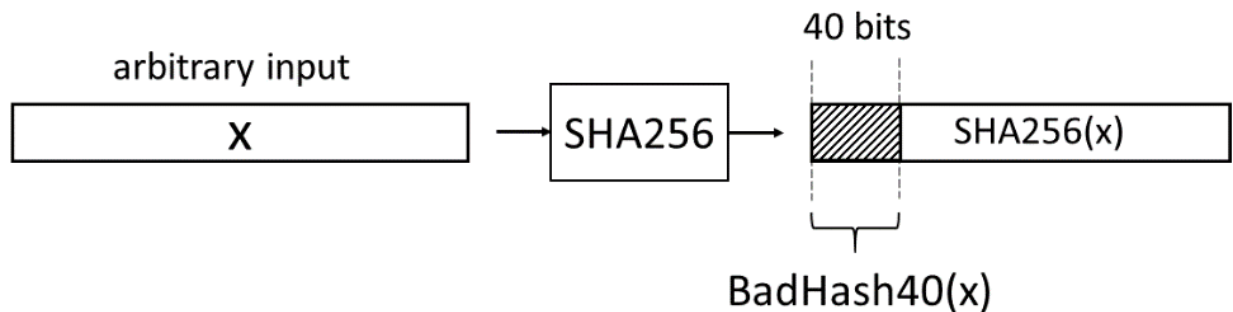


Figure 1: The construction of BadHash40

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.

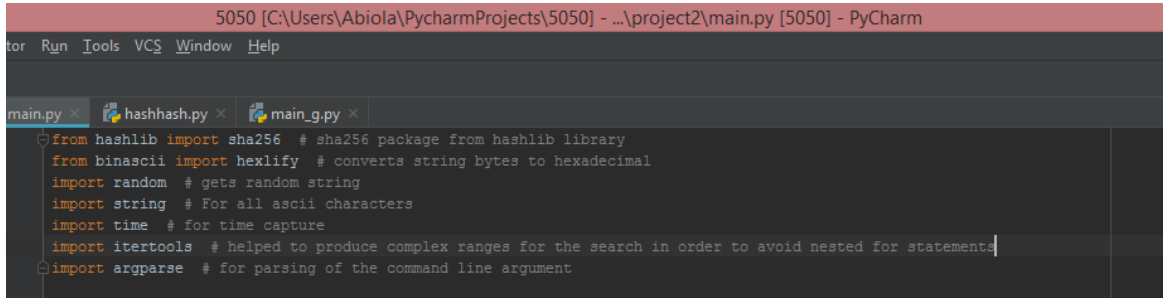
The way the code works is this; random hashes are generated, and the results of each hash are stored as keys in a dictionary (Python's implementation of the hash table data structure). This allows lookup of collisions for already generated hashes to happen in constant time, saving valuable time compared 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



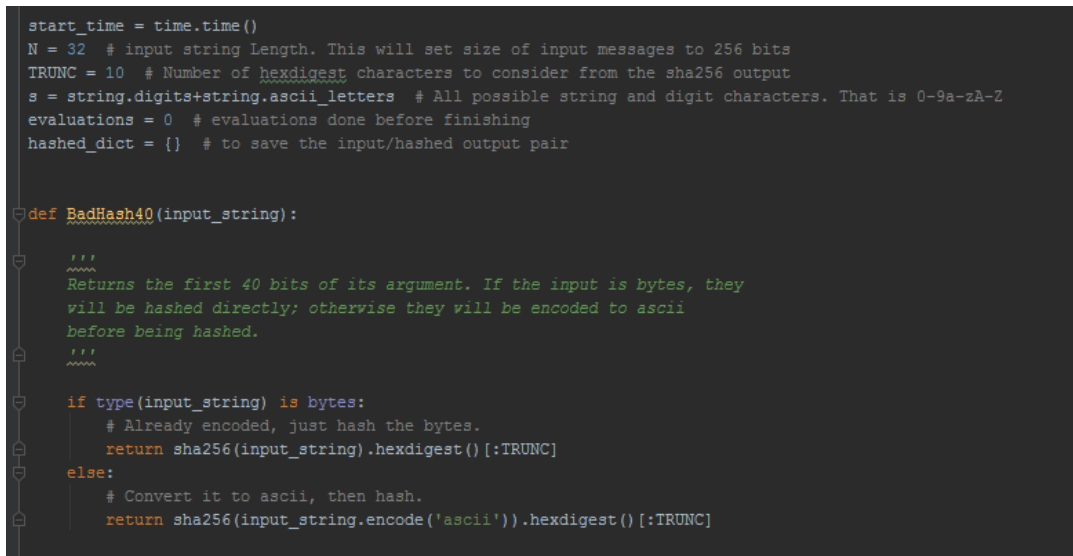
```
5050 [C:\Users\Abiola\PycharmProjects\5050] - ...\project2\main.py [5050] - PyCharm
tor Run Tools VCS Window Help

main.py x hashhash.py x main_g.py x
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 time taken to find a collision, the *itertools* package was used to 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



```
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

```
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

```
'''  
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 generated, 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('-----\n')
    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.

Results

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.

```
terminal
-----Collision Results-----
Number of evaluations made: 453084
Time Taken: 39.91 seconds
Input(in Hex) | Hash
-----+-----
456e4f70466244496179657a34675a6a7848306c376b33545550395358756d77 | e4fed2e2cf
70776a3958343545364d644f54387832627a37667561304a7957597674483352 | e4fed2e2cf
-----+-----

(venv) C:\Users\Abiola\PycharmProjects\5050\project2>
```

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

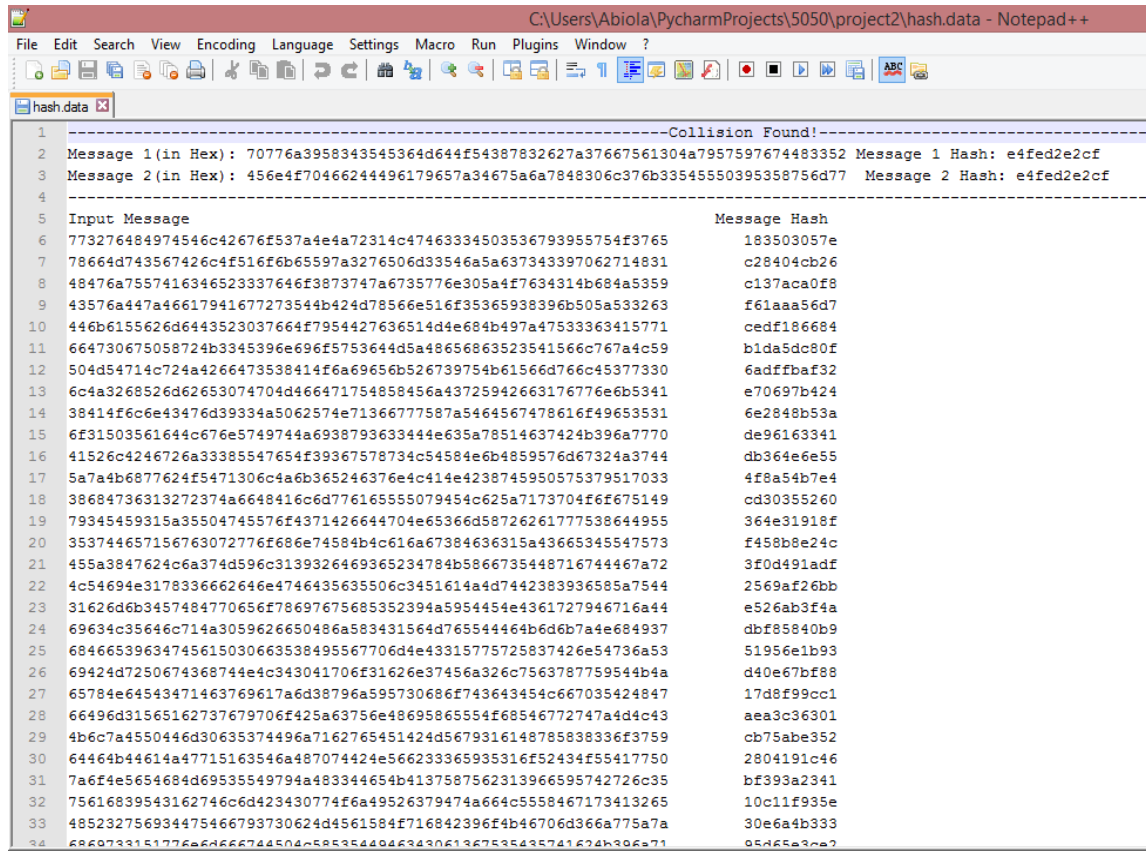
```
aas0376@cse01: ~
-----Collision Results-----
Number of evaluations made: 1184178
Time Taken: 32.74 seconds
Input(in Hex) | Hash
-----+-----
45504a7a625854573239684c67646e596c71355330566f6152514b7047746578 | df16df2c8b
437937344e4c3046666535627a76755474776a584f6d564752634869394d685a | df16df2c8b
-----+-----
aas0376@cse01:~$
```

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

```
aas0376@cse01: ~
-----Collision Results-----
Number of evaluations made: 1043263
Time Taken: 28.83 seconds
Input(in Hex) | Hash
-----+-----
4a434f32794858715a65344c754d6b50695944687433415536354e7249633978 | 46e4aee252
747562376479464a4958416968776c78593076574d314b324f6e473944704e72 | 46e4aee252
-----+-----
aas0376@cse01:~$
```

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.



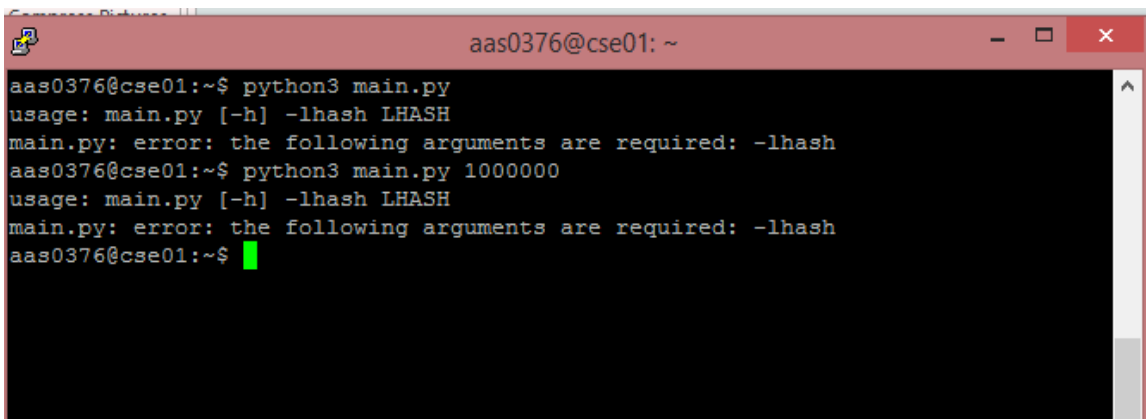
```

1 -----Collision Found!-----
2 Message 1 (in Hex): 70776a3958343545364d644f54387832627a37667561304a7957597674483352 Message 1 Hash: e4fed2e2cf
3 Message 2 (in Hex): 4564ef70466244496179657a34675a6a7848306c376b3354550395358756d77 Message 2 Hash: e4fed2e2cf
4 -----
5 Input Message                                     Message Hash
6 773276484974546c42676f537a4e4a72314c47463334503536793955754f3765 183503057e
7 78664d743567426c4f516f6b65597a3276506d33546a5a637343397062714831 c28404cb26
8 48476a7557416346523337646f3873747a6735776e305a4f7634314b684a5359 c137aca0f8
9 43576a447a46617941677273544b424d78566e516f35365938396b505a533263 f61aaa56d7
10 446b6155626d6443523037664f7954427636514d4e684b497a47533363415771 cedf186684
11 664730675058724b3345396e696f5753644d5a48656863523541566c767a4c59 b1da5dc80f
12 504d54714c724a4266473538414f6a69656b526739754b61566d766c45377330 6adffbfaf32
13 6c4a3268526d62653074704d466471754858456a43725942663176776e6b5341 e70697b424
14 38414f6c6e43476d39334a5062574e71366777587a5464567478616f49653531 6e2848b53a
15 6f31503561644c676e5749744a6938793633444e635a78514637424b396a7770 de96163341
16 41526c4246726a3385547654f39367578734c54584e6b4859576d67324a3744 db364e6e55
17 5a7a4b6877624f5471306c4a6b365246376e4c414e4238745950575379517033 4f8a54b7e4
18 38684736313272374a6648416c6d776165555079454c625a7173704f6f675149 cd30355260
19 79344559315a35504745576f4371426644704e65366d58726261777538644955 364e31918f
20 35374567156763072776f686e74584b4c616a67384636315a43665345547573 f458b8e24c
21 455a3847624c6a374d596c3139326469365234784b5866735448716744467a72 3f0d491adf
22 4c54694e3178336662646e4746435635506c3451614a4d7442383936585a7544 2569af26bb
23 31626d6b3457484770656f78697675685352394a5954454e4361727946716a44 e526ab3f4a
24 69634c35646c714a3059626650486a583431564d765544464b6d6b7a4e684937 dbf85840b9
25 68466539634745615030663538495567706d4e43315775725837426e54736a53 51956e1b93
26 69424d7250674368744e4c343041706f31626e37456a326c7563787759544b4a d40e67bf88
27 65784e64543471463769617a6d38796a595730686f743643454c667035424847 17d8f99cc1
28 66496d3165162737679706f425a63756e48695865554f68546772747a4d4c43 aea3c36301
29 4b6c7a4550446d30635374496a7162765451424d5679316148785838336f3759 cb75abe352
30 64464b44614a47715163546a487074424e56623365935316f52434f55417750 2804191c46
31 7a6f4e5654684d69535549794a483344654b4137587562313966595742726c35 bf393a2341
32 75616839543162746c6d423430774f6a49526379474a664c5558467173413265 10c11f935e
33 485232756934475466793730624d4561584f716842396f4b46706d366a775a7a 30e6a4b333
34 6860733151776e6f667445045853544946343061367535435741624b396e71 9d6f5a3e2

```

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.



```

aas0376@cse01: ~
aas0376@cse01:~$ python3 main.py
usage: main.py [-h] -lhash LHASH
main.py: error: the following arguments are required: -lhash
aas0376@cse01:~$ python3 main.py 1000000
usage: main.py [-h] -lhash LHASH
main.py: error: the following arguments are required: -lhash
aas0376@cse01:~$

```

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>