Jaxon Haws
Jonathan Schreiber
CPE 321, Section 3

Assignment: Hashing and Passwords

1 Task 1

For this task, we explored SHA-256 hashes and created hash collisions for shortened hash bit lengths. In part a, we are just hashing arbitrary inputs. In part b, we are hashing two inputs with a hamming distance of 1. In part C, we are generating hash collisions for differing length SHA-256 hashes. For the program, parts a and b were pretty straight forward and provided expected results. However, for part c is where it got more difficult. We opted to implement the second method of finding hash collisions by relying on the birthday problem. It produces results very quickly, however, it is very memory intensive. In order to produce results, we had to tweak a couple of parameters to prevent it from consuming over 16GB of memory. For the 50-bit hash, it consumes approximately 13GB of memory at its peak.

1.1 Code

```python
from itertools import permutations
from hashlib import sha256
from time import time

def main():
    # Print menu for task
    print("Menu")
    print("a.) Hash arbitrary inputs")
    print("b.) Hash 2 arbitrary inputs whose Hamming Distance is 1 bit")
    print("c.) Hash collisions with 8-50 bits")
    option = input("Enter a choice: ")

    # Execute menu choice
    if (option == 'a'):
        plaintext = input("Enter plaintext to be hashed using SHA256: ")
        sha256hex = create_sha256_hash(plaintext)
        print(sha256hex)
    elif (option == 'b'):
        plaintext1 = input("Enter plaintext to be hashed using SHA256: ")
        plaintext2 = input("Enter plaintext to be hashed using SHA256: ")
        sha256hex1 = create_sha256_hash(plaintext1)
        sha256hex2 = create_sha256_hash(plaintext2)
        print(sha256hex1)
        print(sha256hex2)
    elif (option == 'c'):
        for i in range(8, 51):
            generate_collisions(i)
```

```python

# Returns hex string SHA256 hash
def create_sha256_hash(plaintext: str) -> str:
    hash = sha256(plaintext.encode('utf-8')).hexdigest()
    return hash

# Returns the bitwise representation of SHA256 hash of specified length
def create_sha256_bits(plaintext: str, length: int) -> str:
    hash = create_sha256_hash(plaintext)
    bits = bin(int(hash, base=16)).lstrip('0b')
    return bits[:length]

# Generates a list of characters to use for collision generation
def get_list_of_chars():
    list = []
    for i in range(48, 145):
        list.append(chr(i))
    return list

# Generates collisions for hashes of specific length
def generate_collisions(bit_len: int):
    start_time = time()
    input_count = 0
    # Get a list of legal characters ascii codes 1-255
    legal_chars = get_list_of_chars()

    # Key: raw bits of hash, Value: plaintext string
    htable = {}

    # For strings of length 0 to 8
    for str_len in range(1, 5):

        # Get all permutations of legal characters of length str_length
        perms = list(permutations(legal_chars, r=str_len))

        # For each permutation generated...
        # Merge the tuple into a string
        # Hash the string and get raw bits
        for tup in perms:
            plaintext = ''.join(tup)
            hash_bits = create_sha256_bits(plaintext, bit_len)
            input_count += 1

            # If raw bits exist as a key, we have a collision
            if hash_bits in htable:
                delta = time() - start_time
                print(f"COLLISION at {bit_len} bits: {{{hash_bits:50}}} : "
                f"{{{plaintext.encode('utf-8').hex():10}}} and
    {{{htable[hash_bits].encode('utf-8').hex():10}}} "
                f"in {delta:7.4f} seconds with {input_count} inputs")
                return
            else:
                htable[hash_bits] = plaintext

if __name__ == "__main__":
    main()
```

1.2 Usage

```
$ python3 task1.py
```

1.3 Results

```
Menu
a.) Hash arbitrary inputs
b.) Hash 2 arbitrary inputs whose Hamming Distance is 1 bit
c.) Hash collisions with 8-50 bits
Enter a choice: c
COLLISION at 8 bits: {11100111                                          } :
    {3a        } and {36        } in  0.0004 seconds with 11 inputs
COLLISION at 9 bits: {111001111                                         } :
    {3a        } and {36        } in  0.0002 seconds with 11 inputs
COLLISION at 10 bits: {1100010101                                        }
    : {64        } and {3e        } in  0.0004 seconds with 53 inputs
COLLISION at 11 bits: {11000101011                                       }
    : {64        } and {3e        } in  0.0005 seconds with 53 inputs
COLLISION at 12 bits: {110001010110                                      }
    : {64        } and {3e        } in  0.0005 seconds with 53 inputs
COLLISION at 13 bits: {1101010110101                                     }
    : {3048      } and {c289      } in  0.0057 seconds with 121 inputs
COLLISION at 14 bits: {11010101101011                                    }
    : {3048      } and {c289      } in  0.0040 seconds with 121 inputs
COLLISION at 15 bits: {111111011110011                                   }
    : {30c290    } and {65        } in  0.0038 seconds with 193 inputs
COLLISION at 16 bits: {1111110111100110                                  }
    : {30c290    } and {65        } in  0.0044 seconds with 193 inputs
COLLISION at 17 bits: {11010000000111100                                 }
    : {317b      } and {3065      } in  0.0040 seconds with 268 inputs
COLLISION at 18 bits: {110110011111001011                                }
    : {3255      } and {3159      } in  0.0028 seconds with 326 inputs
COLLISION at 19 bits: {1000010110110111110                               }
    : {3954      } and {376e      } in  0.0050 seconds with 997 inputs
COLLISION at 20 bits: {10000101101101111100                              }
    : {3954      } and {376e      } in  0.0037 seconds with 997 inputs
COLLISION at 21 bits: {100001011011011111001                             }
    : {3954      } and {376e      } in  0.0034 seconds with 997 inputs
COLLISION at 22 bits: {1000010110110111110011                            }
    : {3954      } and {376e      } in  0.0027 seconds with 997 inputs
COLLISION at 23 bits: {10111000110001011100000                           }
    : {4346      } and {50        } in  0.0047 seconds with 1943 inputs
COLLISION at 24 bits: {101110001100010111000001                          }
    : {4346      } and {50        } in  0.0059 seconds with 1943 inputs
```

```
COLLISION at 25 bits: {1011100011000101110000010                         }
    : {4346      } and {50        } in  0.0050 seconds with 1943 inputs
COLLISION at 26 bits: {10100110100101110100010011                        }
    : {303278    } and {4ec286    } in  0.0690 seconds with 9575 inputs
COLLISION at 27 bits: {101110001100010111000001001                       }
    : {306760    } and {50        } in  0.0687 seconds with 14587 inputs
COLLISION at 28 bits: {1011100011000101110000010010                      }
    : {306760    } and {50        } in  0.0664 seconds with 14587 inputs
COLLISION at 29 bits: {11111010011100000000001100111                     }
    : {307538    } and {467e      } in  0.0685 seconds with 15877 inputs
COLLISION at 30 bits: {111010111001001010100010001110                    }
    : {337652    } and {324a6d    } in  0.1080 seconds with 43358 inputs
COLLISION at 31 bits: {1110101110010010101000100011100                   }
    : {337652    } and {324a6d    } in  0.1066 seconds with 43358 inputs
COLLISION at 32 bits: {10010001110010001111100010100000                  }
    : {35c29042  } and {3037c289  } in  0.1374 seconds with 64052 inputs
COLLISION at 33 bits: {101000000100110111110111100100100                 }
    : {3d3c5d    } and {304b5b    } in  0.2351 seconds with 129153 inputs
COLLISION at 34 bits: {1010000001001101111101111001001000                }
    : {3d3c5d    } and {304b5b    } in  0.2343 seconds with 129153 inputs
COLLISION at 35 bits: {10100001000101110010011101001010011               }
    : {414a6f    } and {326350    } in  0.2894 seconds with 166886 inputs
COLLISION at 36 bits: {101000010001011100100111010010100111              }
    : {414a6f    } and {326350    } in  0.2918 seconds with 166886 inputs
COLLISION at 37 bits: {1011011000000001111000110000011101011             }
    : {547f45    } and {3c4257    } in  0.5610 seconds with 345161 inputs
COLLISION at 38 bits: {10011110111101101101010111100000111101            }
    : {5a3945    } and {4bc28e47  } in  0.6290 seconds with 393325 inputs
COLLISION at 39 bits: {100111101111011011010101111000001111011           }
    : {5a3945    } and {4bc28e47  } in  0.6335 seconds with 393325 inputs
COLLISION at 40 bits: {1001111011110110110101011110000011110110          }
    : {5a3945    } and {4bc28e47  } in  0.6364 seconds with 393325 inputs
COLLISION at 41 bits: {10100000000101000110011001010110101011011         }
    : {3157787a  } and {314f426a  } in  8.3014 seconds with 2097321 inputs
COLLISION at 42 bits: {101000000001010001100110010101101010110110        }
    : {3157787a  } and {314f426a  } in  8.2928 seconds with 2097321 inputs
COLLISION at 43 bits: {1010000000010100011001100101011010101101100       }
    : {3157787a  } and {314f426a  } in  8.4101 seconds with 2097321 inputs
COLLISION at 44 bits: {10100000000101000110011001010110101011011001      }
    : {3157787a  } and {314f426a  } in  8.1832 seconds with 2097321 inputs
COLLISION at 45 bits: {100010000111101101110100010100000100101111101     }
    : {39516258  } and {3673c28256} in 19.3647 seconds with 8899880 inputs
COLLISION at 46 bits: {1000100001111011011101000101000001001011111011    }
    : {39516258  } and {3673c28256} in 19.3876 seconds with 8899880 inputs
COLLISION at 47 bits: {10101111010110001101100101000100100110000011010   }
    : {403374c283} and {3f7b5460  } in 28.8664 seconds with 14643604 inputs
COLLISION at 48 bits: {110000100101111110101001100000001110001101000101  }
    : {45785576  } and {42465170  } in 36.5193 seconds with 19534412 inputs
COLLISION at 49 bits: {1110101100011101011111110010111000001101011100000 }
    : {463d6833  } and {3d7e5758  } in 36.6329 seconds with 19875379 inputs
COLLISION at 50 bits: {11101011000111010111111100101110000011010111000001}
    : {463d6833  } and {3d7e5758  } in 36.6303 seconds with 19875379 inputs
```

1.3.1 Graphs

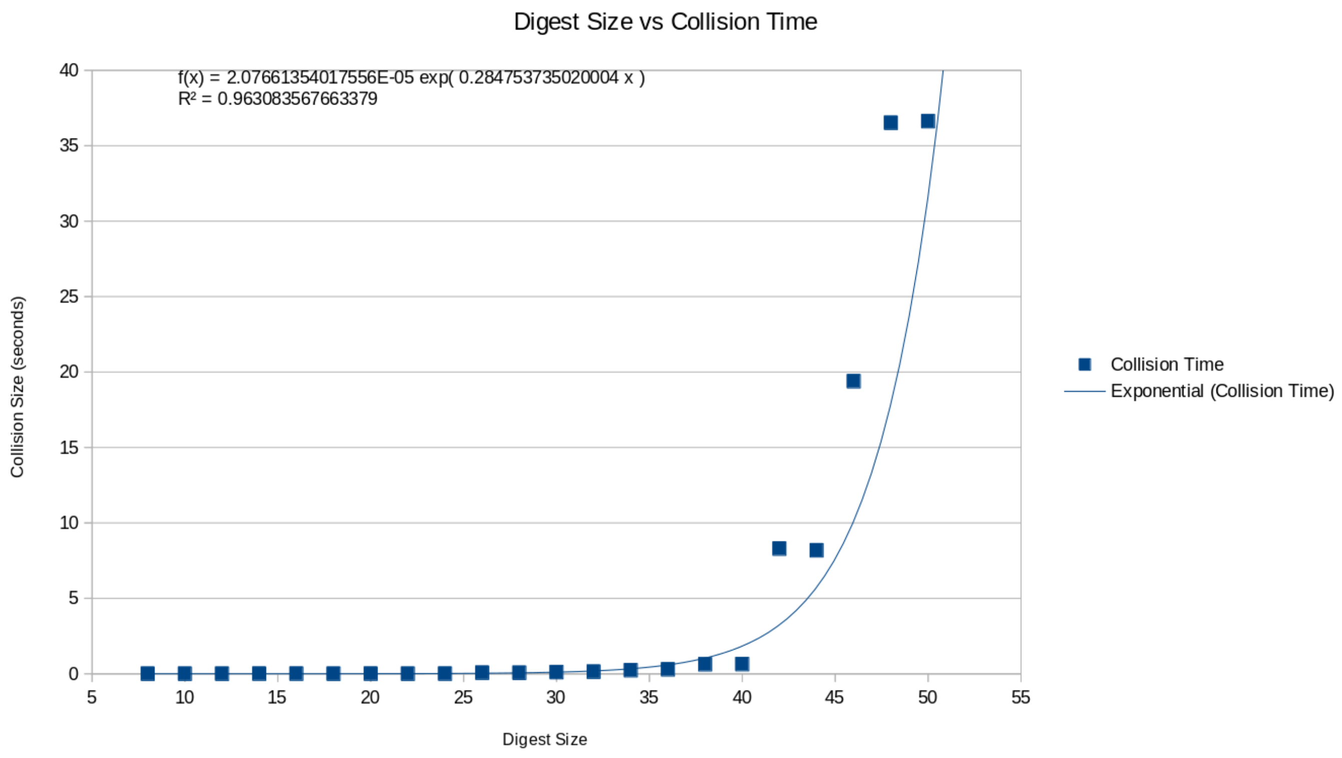

Figure 1: Digest Size and Collision Time

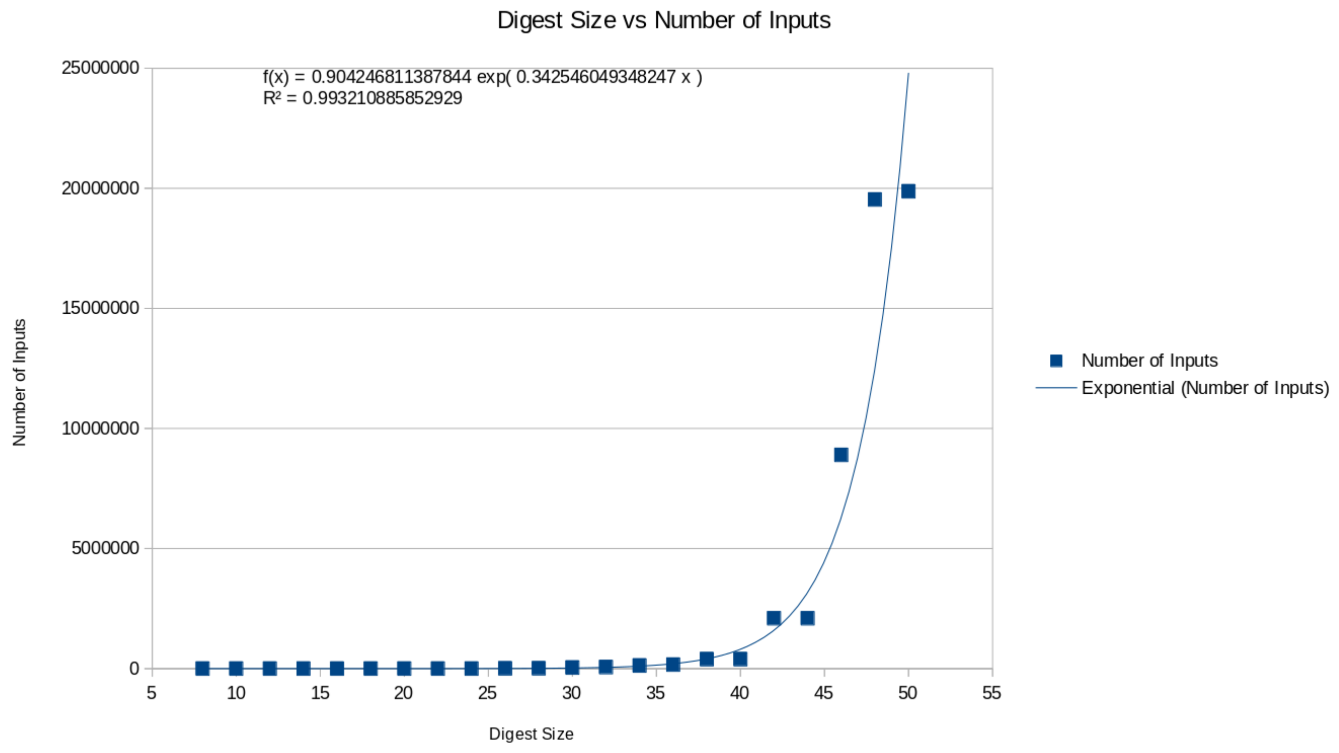

Figure 2: Digest Size and Number of Inputs

2 Task 2

For this task, we are breaking bcrypt password hashes of varying workfactors and salts with known password characteristics. Fortunately, the password length is known to be between 6 and 10 characters and random word from the ntlk word corpus. This produces about 135,000 possible words that can be the password for each bcrypt hash. Additionally, we opted to sort this list based upon length in hopes of shorter passwords being used.

Overall, our script is successful and produced all of the passwords corresponding to the hashes. However, it is a very time intensive task for hashes of workfactor 10+. We made some optimizations, such as cracking passwords simultaneously that share the same salt. Further optimizations we wanted to implement were multiprocessing, especially in relation to CUDA/utilizing a GPU. The total runtime of this script on an Intel i7-8565U (mobile laptop CPU) was approximately 13 hours. With multiprocessing this time could be greatly decreased.

2.1 Code

```python
from bcrypt import hashpw
from nltk.corpus import words
from progressbar import ProgressBar

def main():
    wordlist = get_wordlist()
    user_list = []
    hash_list = []

    with open("shadow.txt", "r") as shadow:
        line = shadow.readline().strip()
        while line != "":
            temp = line.split(":")
            user_list.append(temp[0])
            hash_list.append(temp[1])

            line = shadow.readline().strip()

    passwords = crack_passwords(hash_list, wordlist)

def crack_passwords(hash_list: list, wordlist:list) -> dict:
    password_dict = {}
    for hash in hash_list:
        password_dict[hash] = None

    for hash_str in hash_list:
        hash = hash_str.encode()
        salt_str = hash_str[:29]
        salt = salt_str.encode()
        shared_salts = []
        for h in hash_list:
            if salt_str in h:
                shared_salts.append(h)

        if (password_dict[hash_str] is None):
            pbar = ProgressBar()
            print(f"Cracking shared salts: {shared_salts}")
            for word in pbar(wordlist):
                new_hash = hashpw(word.encode(), salt)
                if (new_hash.decode() in shared_salts):
                    password_dict[new_hash.decode()] = word
                    shared_salts.remove(new_hash.decode())
                    print(f"\n\nHash broken: {{{new_hash.decode()}}} : {{{password_dict[new_hash.decode()]}}}")
                    print(f"Shared salts remaining: {shared_salts}\n")
                if (len(shared_salts) == 0):
                    break

    return password_dict

def get_wordlist() -> list:
    wordlist_full = words.words()
    wordlist = [ n for n in wordlist_full if len(n) >= 6 and len(n) <= 10]
    wordlist.sort(key=len, reverse=False)
    return wordlist

if __name__ == "__main__":
    main()
```

2.2 Supplied Hashes

```
Bilbo:$2b$08$J9FW66ZdPI2nrIMc0xFYI.qx268uZn.ajhymLP/YHaAsfBGP3Fnmq
Gandalf:$2b$08$J9FW66ZdPI2nrIMc0xFYI.q2PW6mqALUl2/uFvV9OFNPmHGNPa6YC
Thorin:$2b$08$J9FW66ZdPI2nrIMc0xFYI.6B7jUcPdnqJz4tIUwKBu8lNMs5NdT9q
Fili:$2b$09$M9xNRFBDn0pUkPKIVCSBzuwNDDNTMWlvn7lezPr8IwVUsJbys3YZm
Kili:$2b$09$M9xNRFBDn0pUkPKIVCSBzuPD2bsU1q8yZPlgSdQXIBILSMCbdE4Im
Balin:$2b$10$xGKjb94iwmlth954hEaw3O3YmtDO/mEFLIO0a0xLK1vL79LA73Gom
Dwalin:$2b$10$xGKjb94iwmlth954hEaw3OFxNMF64erUqDNj6TMMKVDcsETsKK5be
Oin:$2b$10$xGKjb94iwmlth954hEaw3OcXR2H2PRHCgo98mjS11UIrVZLKxyABK
Gloin:$2b$11$/8UByex2ktrWATZOBLZ0DuAXTQl4mWX1hfSjliCvFfGH7w1tX5/3q
Dori:$2b$11$/8UByex2ktrWATZOBLZ0Dub5AmZeqtn7kv/3NCWBrDaRCFahGYyiq
Nori:$2b$11$/8UByex2ktrWATZOBLZ0DuER3Ee1GdP6f30TVIXoEhvhQDwghaU12
Ori:$2b$12$rMeWZtAVcGHLEiDNeKCz8OiERmh0dh8AiNcf7ON3O3P0GWTABKh0O
Bifur:$2b$12$rMeWZtAVcGHLEiDNeKCz8OMoFL0k33O8Lcq33f6AznAZ/cL1LA0yK
Bofur:$2b$12$rMeWZtAVcGHLEiDNeKCz8Ose2KNe821.l2h5eLffzWoP01DlQb72O
Durin:$2b$13$6ypcazOOkUT/a7EwMuIjH.qbdqmHPDAC9B5c37RT9gEw18BX6F0ay
```

2.3 Usage

```
$ pip3 install progressbar nltk
$ python3 task2.py
```

2.4 Results

```
USERNAME:$2b$WORKFACTOR$SALTHASH
    {PASSWORD}   in HH:MM:SS % of wordlist hashed
------------------------------------------------------------------
Bilbo:$2b$08$J9FW66ZdPI2nrIMcOxFYI.qx268uZn.ajhymLP/YHaAsfBGP3Fnmq
    {welcome}    in 00:08:18 30%
Gandalf:$2b$08$J9FW66ZdPI2nrIMcOxFYI.q2PW6mqALUl2/uFvV9OFNPmHGNPa6YC
    {wizard}     in 00:03:31 12%
Thorin:$2b$08$J9FW66ZdPI2nrIMcOxFYI.6B7jUcPdnqJz4tIUwKBu8lNMs5NdT9q
    {diamond}    in 00:04:48 17%
------------------------------------------------------------------
Fili:$2b$09$M9xNRFBDn0pUkPKIVCSBzuwNDDNTMWlvn7lezPr8IwVUsJbys3YZm
    {desire}     in 00:01:45 3%
Kili:$2b$09$M9xNRFBDn0pUkPKIVCSBzuPD2bsU1q8yZPlgSdQXIBILSMCbdE4Im
    {ossify}     in 00:04:15 7%
------------------------------------------------------------------
Balin:$2b$10$xGKjb94iwmlth954hEaw3O3YmtDO/mEFLIO0a0xLK1vL79LA73Gom
    {hangout}    in 00:24:29 20%
Dwalin:$2b$10$xGKjb94iwmlth954hEaw3OFxNMF64erUqDNj6TMMKVDcsETsKK5be
    {drossy}     in 00:03:49 3%
Oin:$2b$10$xGKjb94iwmlth954hEaw3OcXR2H2PRHCgo98mjS11UIrVZLKxyABK
    {ispaghul}   in 00:52:06 40%
------------------------------------------------------------------
Gloin:$2b$11$/8UByex2ktrWATZOBLZ0DuAXTQl4mWX1hfSjliCvFfGH7w1tX5/3q
    {oversave}   in 01:58:14 43%
Dori:$2b$11$/8UByex2ktrWATZOBLZ0Dub5AmZeqtn7kv/3NCWBrDaRCFahGYyiq
    {indoxylic}  in 02:28:22 62%
Nori:$2b$11$/8UByex2ktrWATZOBLZ0DuER3Ee1GdP6f30TVIXoEhvhQDwghaU12
    {swagsman}   in 01:59:01 49%
Ori:$2b$12$rMeWZtAVcGHLEiDNeKCz8OiERmh0dh8AiNcf7ON3O3P0GWTABKh0O
    {airway}     in 00:01:08 0.3%
Bifur:$2b$12$rMeWZtAVcGHLEiDNeKCz8OMoFL0k33O8Lcq33f6AznAZ/cL1LAOyK
    {corrosible} in 05:52:18 81%
Bofur:$2b$12$rMeWZtAVcGHLEiDNeKCz8Ose2KNe821.l2h5eLffzWoP01DlQb72O
    {libellate}  in 04:37:00 63%
------------------------------------------------------------------
Durin:$2b$13$6ypcazOOkUT/a7EwMuIjH.qbdqmHPDAC9B5c37RT9gEw18BX6FOay
    {purrone}    in 03:50:24 25%
```

3 Questions

3.1 What did you observe based on Task 1b? How many bytes are different between two digests?

In Task 1b, with inputs that had a hamming distance of 1, we noticed that the digests that were produced were drastically different. It seemed that all 32 bytes were different with no way of telling that the two inputs were similar.

3.2 What is the maximum number of files you would ever need to hash to find a collision of an n-bit digest? Given the birthday bound, what is the expected number of hashes before a collision on an n-bit digest? Is this what you observed? Based on the data you have collected, speculate on how long it might take to find a collision on the full 256-bit digest.

In order to find a hash collision of an n-bit digest, you would need need a maximum of $2^n + 1$ files. Given the birthday bound, you would also need $2^n + 1$ hashes before a collision of an n-bit digest. In our case, it was often much less to achieve a collision. If we use the equation on our graph, it would take approximately $1.0974 * 10^{38}$ different values to find a hash collision. This is much smaller than $2^{256} + 1$ inputs that are required to find a collision. At a max rate of 8192.0 MH/s on an Nvidia GTX 3080 (according to the hashcat forums), it would still take about $1.3395 * 10^{28}$ seconds, or about $4.249 * 10^{20}$ years.

3.3 Given an 8-bit digest, would you be able to break the one-way property (i.e. can you find any pre-image)? Do you think this would be easier or harder than finding a collision? Why or why not?

Given an 8-bit digest, you could easily break the one-way property of SHA-256. This is technically harder to do than just finding a collision due to the nature of hashing, however. If you were just trying to find any collision you could systematically hash $2^8 + 1$ different values until one of them yields in a collision. This is different from finding a pre-image for a specific digest due to the collision resistant nature of SHA-256. While it likely wouldn't take much more than $2^8 + 1$ different inputs, it could take exponentially more if you are unlucky.

3.4 For Task 2, given your results, how long would it take to brute force a password that uses the format word1:word2 where both words are between 6 and 10 character? What about word1:word2:word3? What about word1:word2:number where number is between 1 and 5 digits? Make sure to sufficiently justify your answers.

Using the nltk corpus as our wordlist, if the format of the password was word1:word2, we would have approximately $135,000 * 135,000 = 18,225,000,000$ different possible combinations. Assuming a workfactor of 8 and same amount of compute power and single threaded program, it would take approximately $8 * 18,225,000,000/135,000 = 1,080,000$ minutes, or approximately 2.05 years. However, if you were to parallelize the hashing with use of a GPU, this time could be drastically reduced, to the point of being possible in a reasonable amount of time.

Using the nltk corpus as our wordlist, if the format of the password was word1:word2:word3, we would have approximately $135,000 * 135,000 * 135,000 = 2.46 * 10^{15}$ different possible combinations. Assuming a workfactor of 8 and same amount of compute power and single threaded program, it would take approximately $8 * 2.46 * 10^{15}/135,000 = 145,800,000,000$ minutes, or approximately 277,397 years. However, if you were to parallelize the hashing with use of a GPU, this time could be drastically reduced.

Using the nltk corpus as our wordlist, if the format of the password was word1:word2:number, we would have approximately $135,000 * 135,000 * 5 = 91,125,000,000$ different possible combinations. Assuming a workfactor of 8 and same amount of compute power and single threaded program, it would take approximately $8 * 91,125,000,000/135,000 = 5,400,000$ minutes, or approximately 10.27 years. However, if you were to parallelize the hashing with use of a GPU, this time could be drastically reduced, to the point of being possible in a more reasonable amount of time.