Authentication and Hash

Kyle Jennings, Zarek Lazowski, Evan Morris

Questions

1. What do you observed based on Task 1b? How many bytes are different between the two digests?
    a. The two digests are pretty much completely different. As far as I can tell most if not all the bytes are completely different.
2. What is the maximum number of files you would ever need to hash to find a collision on an n-bit digest? Given the birthday bound, what is the expected number of hashes before a collision on a 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.
    a. The maximum number for an n-bit digest is 2^n. So, a 10-bit digest would have 2^10 or 1024 unique values. To guarantee a collision you would do the unique values + 1 (1024 + 1).
    b. There is an equation to figure out the number of items to check to get a 50% chance of finding a collision.

$$n \cong \frac{1}{2} + \sqrt{\frac{1}{4} + 2 * \ln(2) * 2^n}$$

    So for a 10-bit digest we would plug n = 10 into the above equation and get 38.1 as our output. For our data we got 23, which is kind of close.

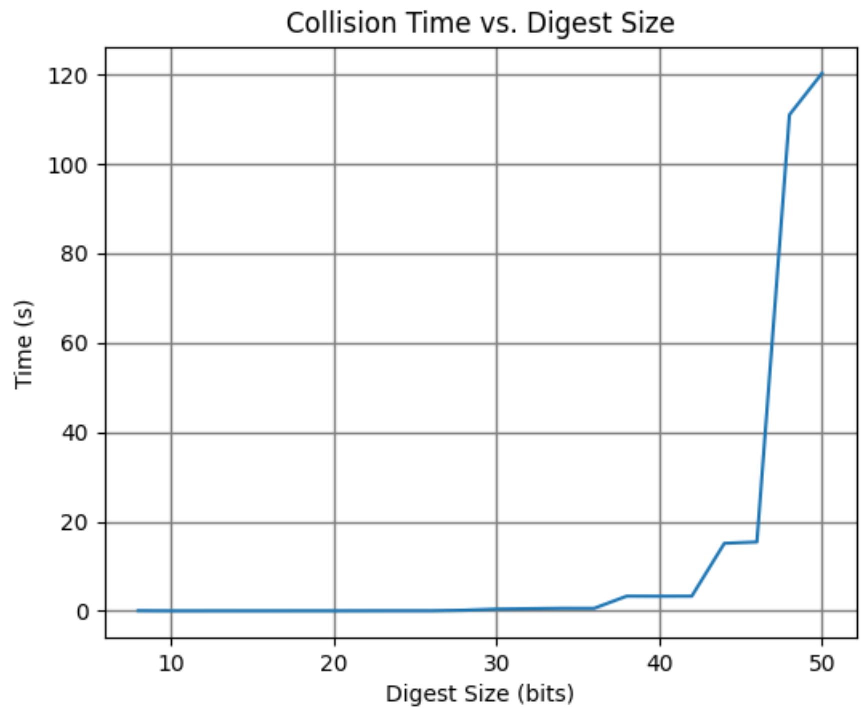
    For a 30-bit digest we would plug n = 30 into the above equation and get 38582 as our output. For our data we got 65856. Again, not super close but this is because we are calculating for 50% and we may or may not get lucky in the actual results.
    c. For a 256-bit hash we could plug that into the equation, and we would get 4*10^38. Very large.
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?
    a. Yes, there are only 2^8 (256) possible digests, so any operation would be trivial.
    b. This is more difficult than finding a collision because you have to check one value against all the other values instead of all the values against each other.

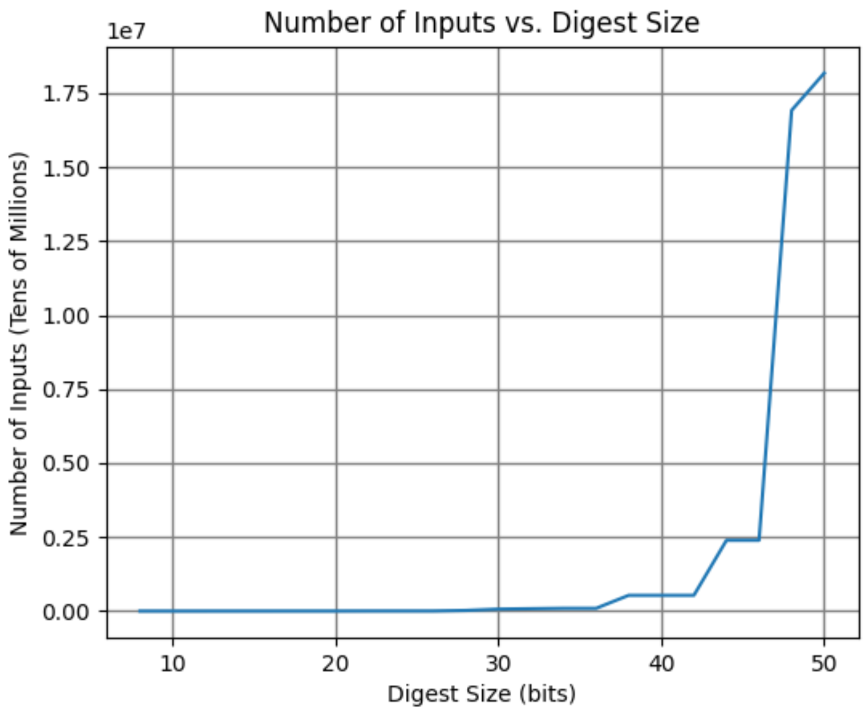
4. For Task 2, given your results, how long would it take to brute force a password that uses the format word1:word2 where both words are between 6 and 10 characters? What about word1:word2:word3? What about word1:word2:number where number is between 1 and 5 digits? Make sure to sufficiently justify your answers.
    a. The number of words in the corpus is around 135,000 words. With only word1 we only must check the 135,000 values. Every time we add another word we have to increase the exponent that value is raised to (135,000^1..2..3..etc.). This would effectively square the original time with one extra word and cube it with two.
    b. word1:word2 -> (135,000^2) values to search through
    c. word1:word2:word3 -> (135,000^3) values to search through
    d. As far as adding a number at the end goes, we only have to multiply the previous value by the number of possible numbers, in this case, with 5 digits, we would have to multiply it by 10^5 (100000).
    e. word1:word2:number -> (135,000^2 * 10^5)

Task 1

The first task had two main parts. The first was generating a set of bytes that is a Hamming Distance of one away from another set of bytes and comparing the resulting hashes of those two functions. To do this we start with any byte string, we used “Hello, world!”. Then we convert that byte string to an integer using the *int.from_bytes()* function. Finally, we pick ten random numbers within the bit length of the original byte string (“Hello, world!” is 104 bits long) and XOR that bit with 1, flipping it. If we do this ten time, we get ten unique byte combinations that are 1 bit off the original. Then we just run all these byte arrays through the hash function and compare their outputs. None of them were similar.

The second part of task 1 asks us to find a hash collision for different digest lengths and plot the data. To do this we convert every number from 0 to 2^n for each bit length to a string and then insert that hash of that string into a dictionary as a key. Every time we insert one of these digests we can check to see if the key exists already, and if it does, we found a collision! Now just repeat that for every bit length between 8 and 50 bits. After that a little bit of matplotlib helps with the plotting of the data.


Collision Time vs. Digest Size
Time (s)
0
20
40
60
80
100
120
10
20
30
40
50
Digest Size (bits)


Number of Inputs vs. Digest Size
1e7
Number of Inputs (Tens of Millions)
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
10
20
30
40
50
Digest Size (bits)

Task 2

Task 2 is more complicated. First, we need some setup. To get the list of possible passwords we need to import the “words” corpus from nltk and filter it to only have words between 6 and 10 characters. Then we must read all the inputs from the shadow.txt file and split all the salted passwords into the salt by itself and the entirety of the salted password.

For our implementation we used python’s multiprocessing to speed up the program. The way we accomplished this was by setting up a worker function that checks a single word against the password we want to crack. This way we can work on 10 words at the same time (or however many processors you have). Then we just check every single word in the input set against all the salted passwords.

Because the program takes so long to run, we also added a loop that keeps track of how many words have been checked and compares it to how many are left to get a time estimate.

As we continue through the list of passwords the hashing function takes more and more time to run. At the beginning of the program (for the first few passwords) the program can process almost 600 words per second. For the last few passwords this number dropped down to something like 20 per second. This was expected because the hash gets much more rigorous as we move down the list of passwords.

Passwords

The password for user [Bilbo] is [welcome]
The password for user [Gandalf] is [wizard]
The password for user [Thorin] is [diamond]
The password for user [Fili] is [desire]
The password for user [Kili] is [ossify]
The password for user [Balin] is [hangout]
The password for user [Dwalin] is [drossy]
The password for user [Oin] is [ispaghul]
The password for user [Gloin] is [oversave]
The password for user [Dori] is [indoxylic]
The password for user [Nori] is [swagsman]
The password for user [Ori] is [airway]
The password for user [Bifur] is [corrosible]
The password for user [Bofur] is [libellate]
The password for user [Durin] is [purrone]

util.py
```python
from cryptography.hazmat.primitives import hashes


def string_SHA256(b: bytes):
    digest = hashes.Hash(hashes.SHA256())
    digest.update(b)
    return digest.finalize().hex()


def short_hash(b: bytes, n: int):
    if not (8 <= n <= 50):
        raise ValueError(f"Invalid hash length {n}, must be in
[8, 50]")

    digest = hashes.Hash(hashes.SHA256())
    digest.update(b)
    hashed = digest.finalize()
    return int(bin(int.from_bytes(hashed, "little"))[2:2+n])
```

task1.py
```python
import csv
import random
import time
from util import *
import matplotlib.pyplot as plt
from os.path import exists
import pandas as pd


def _int_to_bytes(n: int, l: int):
    """Return the bytes representation of `n` with length `l`

    Args:
        n (int): The integer to convert to bytes
        l (int): The length of the bytes to return

    Returns:
        bytes: The bytes representation of `n`
    """
    return n.to_bytes(l // 8, 'little')


def _find_collisions(min=8, max=50) -> tuple[int, int, int, int,
int]:
    """Find collisions for hash lengths between `min` and `max`

    Args:
        min (int, optional): smallest length in bits. Defaults
to 8.
        max (int, optional): longest length in bits. Defaults to
50.

    Yields:
        tuple[int, int, int, int, int]: length, item1, item2,
hash, time_diff
    """
    for bitlen in range(min, max+2, 2):
        hash_to_str = {}
        tic = time.time()
        for i in range(2 ** bitlen):
            i_hash = short_hash(str(i).encode(), bitlen)
            if i_hash in hash_to_str:
                yield bitlen, i, hash_to_str[i_hash], i_hash,
time.time() - tic
```

```
                break
            hash_to_str[i_hash] = i


def task1_b():
    # start with a random byte string
    init_bytes = b"Hello, world!"
    bit_len = 8 * len(init_bytes)
    bits0 = int.from_bytes(init_bytes, 'little')

    # generate a bunch of byte strings with a hamming distance
of 1 from `init_bytes`
    perms = []
    for _ in random.sample(range(bit_len), 10):
        bitnum = random.randint(0, bit_len)
        bits1 = bits0 ^ (1 << bitnum)
        perms.append(_int_to_bytes(bits1, bit_len))


    print("Checking 10 pairs of bytes with hamming distances of
1:\n")
    for perm in perms:

print(f"\t{string_SHA256(init_bytes)}\n\t{string_SHA256(perm)}\n
")


def _get_all_collisions():
    # go through all the lengths and find the collisions
    with open("data/task1.csv", "w") as f:
        cols = ["Length", "Item1", "Item2", "Hash",
"NumChecked", "Time"]
        rows = [cols]
        for bitlen, item2, item1, hash, time_diff in
_find_collisions(min=8, max=50):
            rows.append([bitlen, item1, item2, hash, item2+1,
time_diff])

            print(f"Done with {bitlen} in {time_diff:.2f} s")

        csv.writer(f).writerows(rows)


def task1_c():
    if not exists("data/task1.csv"):
        _get_all_collisions()
```

```python
    with open("data/task1.csv") as f:
        df = pd.read_csv(f)

    # plot the results
    _, axis = plt.subplots(1, 2)

    axis[0].grid(color='gray', linestyle='-', linewidth=1)
    axis[0].plot('Length', 'Time', data=df)
    axis[0].set_title('Collision Time vs. Digest Size')
    axis[0].set_xlabel('Digest Size (bits)')
    axis[0].set_ylabel('Time (s)')

    axis[1].grid(color='gray', linestyle='-', linewidth=1)
    axis[1].plot('Length', 'NumChecked', data=df)
    axis[1].set_title('Number of Inputs vs. Digest Size')
    axis[1].set_xlabel('Digest Size (bits)')
    axis[1].set_ylabel('Number of Inputs (Tens of Millions)')

    plt.ticklabel_format(useOffset=False)
    plt.show()

if __name__ == "__main__":
    task1_b()
    # task1_c()
```

task2.py

```
import csv
import time
import bcrypt as bc
from nltk.corpus import words
import multiprocessing as mp

# get all the words in the nltk corpus that are 6 to 10
characters long
SIX_TO_TEN = [word.encode() for word in words.words() if 6 <=
len(word) <= 10]


def read_input(filename: str) -> dict[str, str]:
    """Reads the input, probably from `shadow.txt` and makes it
a dictionary

    Args:
        filename (str): name of the file to read

    Returns:
        dict[str, str]: dictionary that maps users to the salt
and password
    """
    with open(filename, "rb") as f:
        users = {}
        for line in f.readlines():
            # split each line into the user and the hash
            user, hash = line.strip().split(b":")
            # save the salt and the whole thing separately
            users[user] = (hash[:29], hash)
    return users


def init_worker(pi, t, ts, f, d) -> None:
    """Workers need to start with these values

    Args:
        pi (mp.Value): this is where the correct password should
be saved
        t (bytes): the result of salting and hashing the
password
        ts (bytes): the salt for the password
        f (mp.Event): if the word was found, so we can end early
        d (mp.Event): flag to indicate that no more looking
needs to be done
    """
    global plain_index, target, target_salt, found, done
    plain_index, target, target_salt, found, done = pi, t, ts,
f, d


def worker(num: int, word: str) -> None:
    """Checks a single word against the salted and hashed
password

    Args:
        num (int): index in SIX_TO_TEN
        word (str): the word
    """
    if not done.is_set():
        if bc.hashpw(word, target_salt) == target:
            plain_index.value = num
            found.set()


def main():
    # read the info from the input file
    hashed_pwds = read_input("data/shadow.txt")
    plain_pwds = {user: None for user in hashed_pwds}

    for user, hash in hashed_pwds.items():
        # start a timer
        tic = time.time()
        salt, whole = hash

        # initialize the variables for the workers
        m = mp.Manager()
        found = m.Event()
        done = m.Event()
        plain_index = m.Value("i", -1)

        with mp.Pool(initializer=init_worker,
initargs=(plain_index, whole, salt, found, done)) as p:
            results = {}
            # check every single word in the set
            for i, word in enumerate(SIX_TO_TEN):
                results[i] = p.apply_async(worker, (i, word))
            # every second give a status update as long as we
didn't find the
            # password
            while not found.wait(timeout=1):
                running, successful, error = 0, 0, 0
                toc = time.time()
                for key, result in results.items():
                    try:
                        if result.successful():
                            successful += 1
                        else:
                            error += 1
                    except ValueError:
                        running += 1
                rate = (successful + error) / (toc - tic)

                # display progress on a line
                print(''.join(' ' for _ in range(200)),
end='\r')
                print(f'Running: {running}', end=' ')
                print(f'Successful: {successful}', end=' ')
                print(f'Error: {error}', end=' ')
                print(f'Rate: {rate:.0f}', end=' ')
                print(f'Estimated completion time: \
                    {time.strftime("%H:%M:%S",
time.gmtime(running / rate))}', \
                        end='\r')

            # if we found the password, we can stop looking
            print()
            done.set()

        # save the password if we found it
        if plain_index.value > 0:
            print(f"The password for user
[{user.decode('utf8')}] is
[{SIX_TO_TEN[plain_index.value].decode('utf8')}]")
            plain_pwds[user] = SIX_TO_TEN[plain_index.value]
        else:
            plain_pwds[user] = None
            print(f"Didn't find a password for user
[{user.decode('utf8')}]")

    with open("data/task2.csv", "wb", newline='') as f:
        w = csv.DictWriter(f)
        w.fieldnames(plain_pwds.keys())
        w.writerow(plain_pwds)


if __name__ == "__main__":
    print(len(SIX_TO_TEN))
```