[CSC 321] Asgn 3: Hashing & Passwords

Charlie Ray, Seth Johnson, Kylie ODonnel

July 2024

Task 1

```
from Crypto import Random
import hashlib
import time
import random
import string
import matplotlib.pyplot as plt


## PART A
def sha256_hash(input_string, truncate_bits=256):
    sha256 = hashlib.sha256()
    sha256.update(input_string.encode('utf-8'))
    digest = sha256.digest()
    # Truncate the digest to the specified number of bits
    truncated_digest = digest[:truncate_bits // 8]  # get the first N bytes
    return truncated_digest.hex()


## PART B
def flip_bit(input_string, bit_position):
    byte_array = bytearray(input_string.encode('utf-8'))
    byte_index = bit_position // 8
    bit_index = bit_position % 8
    byte_array[byte_index] ^= 1 << bit_index
    return byte_array.decode('utf-8', errors='ignore')


original_string = "Hello, world!"
hashes = []

# Flip a bit and hash the new string a few times
for i in range(5):
```

```
    modified_string = flip_bit(original_string, i)
    hash_original = sha256_hash(original_string)
    hash_modified = sha256_hash(modified_string)
    hashes.append((original_string, hash_original, modified_string, hash_modified))

# Print the results
for original, hash_orig, modified, hash_mod in hashes:
    print(f"Original string: '{original}'")
    print(f"SHA-256: {hash_orig}")
    print(f"Modified string: '{modified}'")
    print(f"SHA-256: {hash_mod}")
    print("="*60)

## PART C

def random_string(length=10):
    letters = string.ascii_letters + string.digits
    return ''.join(random.choice(letters) for _ in range(length))

def find_collision(bits):
    hash_dict = {}
    attempts = 0
    start_time = time.time()
    while True:
        attempts += 1
        input_string = random_string()
        truncated_hash = sha256_hash(input_string, bits)
        if truncated_hash in hash_dict:
            collision_time = time.time() - start_time
            return hash_dict[truncated_hash], input_string, attempts, collision_time
        else:
            hash_dict[truncated_hash] = input_string

digest_sizes = list(range(8, 52, 2))
results = []
for bits in digest_sizes:
    print(f"Finding collision for {bits}-bit digest...")
    m0, m1, num_attempts, time_taken = find_collision(bits)
    results.append((bits, num_attempts, time_taken))
    print(f"Collision found for {bits}-bit digest:")
    print(f"Message 1: {m0}")
    print(f"Message 2: {m1}")
    print(f"Number of attempts: {num_attempts}")
    print(f"Time taken: {time_taken:.2f} seconds")
    print("="*60)
```

```
bits, attempts, times = zip(*results)

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(bits, attempts, marker='o')
plt.title('Number of Inputs vs Digest Size')
plt.xlabel('Digest Size (bits)')
plt.ylabel('Number of Inputs')

plt.subplot(1, 2, 2)
plt.plot(bits, times, marker='o')
plt.title('Time Taken vs Digest Size')
plt.xlabel('Digest Size (bits)')
plt.ylabel('Time Taken (seconds)')

plt.tight_layout()
plt.show()
```

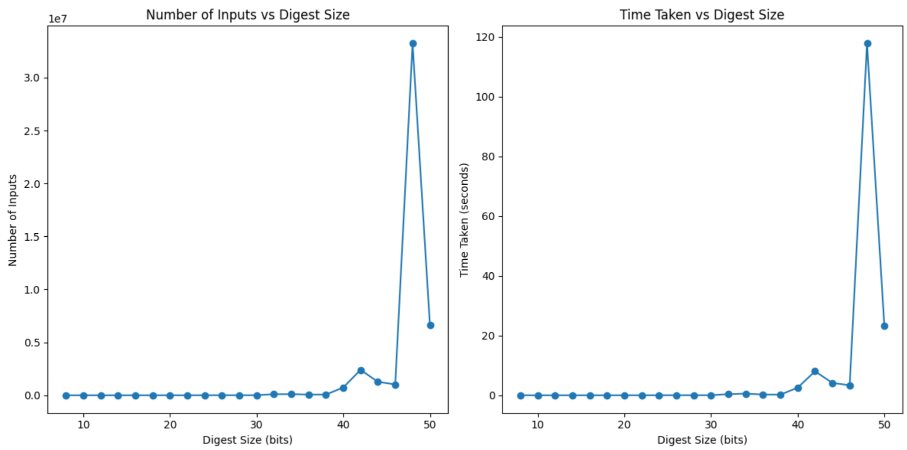
Output from Part B

```
Original string: 'Hello, world!'
SHA-256: 315f5bdb76d078c43b8ac0064e4a0164612b1fce77c869345bfc94c75894edd3
Modified string: 'Iello, world!'
SHA-256: cd5b925a2796adf5b2ba555960c351a78a54d786eb3c2702f67ef2ea81b3f31a
============================================================
Original string: 'Hello, world!'
SHA-256: 315f5bdb76d078c43b8ac0064e4a0164612b1fce77c869345bfc94c75894edd3
Modified string: 'Jello, world!'
SHA-256: eca40e9fe2ecff00b33cd14c6fc46c40de65f9db3e40d1c723720ccb7f6ea70c
============================================================
Original string: 'Hello, world!'
SHA-256: 315f5bdb76d078c43b8ac0064e4a0164612b1fce77c869345bfc94c75894edd3
Modified string: 'Lello, world!'
SHA-256: f4140eab5e47d08d75d0c2d8b65b59b32593dd2672a25edb3e6e578f9a82ae4e
============================================================
Original string: 'Hello, world!'
SHA-256: 315f5bdb76d078c43b8ac0064e4a0164612b1fce77c869345bfc94c75894edd3
Modified string: '@ello, world!'
SHA-256: 34db9e2fcc24e08a2a27084641b34f6376ba6b36b37d6d4b1936943b522f85c0
============================================================
Original string: 'Hello, world!'
SHA-256: 315f5bdb76d078c43b8ac0064e4a0164612b1fce77c869345bfc94c75894edd3
Modified string: 'Xello, world!'
SHA-256: c0ba54382636519e5babda979b3795552620bd7a243a0c599e52770d4d802473
============================================================
```

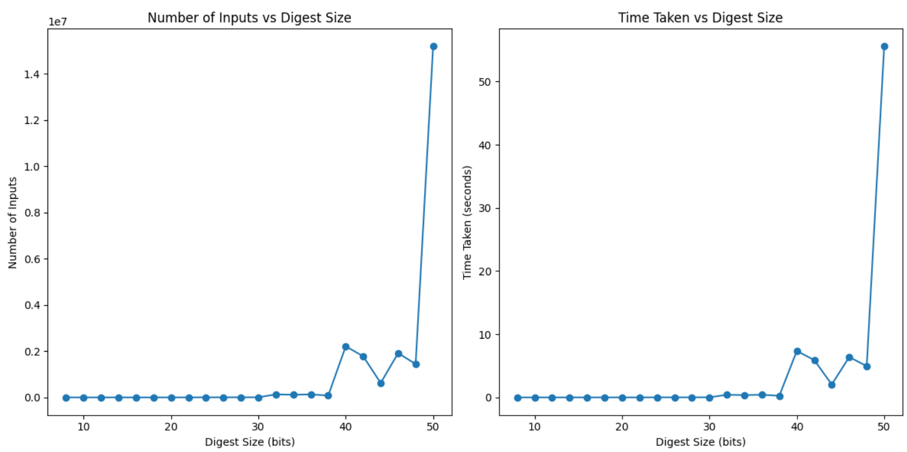
This output clearly demonstrates the avalanche effect of the hashing algorithm, as even a minor change in the input string leads to a large change in the output hash.

Output from part C

First Run:



Second Run:



These figures demonstrate how it takes significantly more inputs and time to find conflicts when the digest size grows larger, as there are more bits that must align. However, as this is a random process, there is variability, which is shown in the unusually high time it took to find a conflict in the first run for the 48 bit digest.

Question 1: What is the maximum number of files you would ever need to hash to find a collision on an n-bit digest?

On an n-bit digest, there are 2^n total possible hashes, thus to guarantee a collision, you would need to hash $2^n + 1$ files, by the pigeonhole principle.

Question 2: Given the Birthday Bound, what is the expected number of hashes before a collision on an n-bit digest? Is this what you observed?

Given the birthday bound, the expected number of hashes before a collision on an n-bit digest would be $2^{\frac{n}{2}}$ or $\sqrt{2^n}$, as this gives a 50% probability of a collision. The following table contains our data.

Digest Size (bits)	Experimental inputs	Theoretical Inputs
8	28	16
10	29	32
12	35	64
14	16	128
16	307	256
18	179	512
20	346	1024
22	183	2048
24	896	4096
26	5404	8196
28	3210	16384
30	1082	32768
32	83034	65536
34	113247	131072
36	86121	262144
38	96864	524288
40	2066662	1048576
42	502090	2097152
44	2621163	4194304
46	361363	8388608
48	35929782	16777216
50	26590800	33554432

Table 1: Collision Attempts and Times for Different Digest Sizes

Our data aligns well with the theoretical numbers, however we did seem to get lucky more often than not.

Question 3: Given the data you've collected, speculate on how long it might take to find a collision on the full 256-bit digest.

From our data, we were able to calculate 26590800 hashes in 217.55 seconds, which is a hash rate of 122000 per second, given that the estimated number of hashes for 256 bits is 2^{128}, it would take $\frac{2^{128}}{122000} = 2.7891997289 * 10^{33}$ seconds, or $8.8444943203 * 10^{25}$ years, which is infeasible.

Question 4: Given an 8-bit digest, would you be able to break the one-way property (i.e. can you find any pre-image)?

Yes, as there are only 2^8 possible combinations, it would be relatively easy to calculate all possible hashes and break it through brute force.

Question 5: Do you think this would be easier or harder (i.e. more or less work) than finding a collision? Why or why not?

It would be harder, as you would have to find the whole output and input space. With collisions you can just get lucky, and then you can stop, but to break the one-way property, you would have to calculate all possible collisions, rather than just one.

Task 2

```
import nltk
from nltk.corpus import words
from bcrypt import *
import time
import threading

# Download and filter the word list
nltk.download('words')
word_list = words.words()
corpus = [word for word in word_list if 6 <= len(word) <= 10]

# Read the shadow file and extract salts
with open("shadow.txt", "r") as shadow_file:
    shadow_lines = shadow_file.readlines()

# Extract salts
salts = [password.split(":")[1].strip() for password in shadow_lines]
```

```
# Split corpus into chunks
chunk_size = 16384
corpi = [corpus[i:i + chunk_size] for i in range(0, len(corpus), chunk_size)]

# Create a threading event to signal when a password is found
password_found_event = threading.Event()

def check_passwords(corpus_chunk, salt, password_file, start_time):
    log_scale = 10
    for index, word in enumerate(corpus_chunk):
        if password_found_event.is_set():
            return  # Stop processing if the password is found in another thread
        if checkpw(word.encode(), salt.encode()):
            print(f"Password for {password_file} is {word}")
            # Write to file once a password is found
            with open(password_file, "a") as file:  # Use "a" mode to append
                file.write(f"{word}\n")
            password_found_event.set()  # Signal other threads to stop
            break
        if index % log_scale == 0:
            elapsed_time = time.time() - start_time
            if index == 0:
                time_remaining = 0
            else:
                time_remaining = elapsed_time / index * (len(corpus_chunk) - index)
            print(f"Attempted {index} passwords, elapsed time: {elapsed_time:.2f}s, time rem
            log_scale *= 2

# Use threading for parallel processing
threads = []
for password in shadow_lines:
    password_file = password.split(":")[0] + ".txt"
    first_salt = password.split(":")[1].strip()
    start_time = time.time()
    password_found_event.clear()  # Reset the event for each new password
    for corpus_chunk in corpi:
        thread = threading.Thread(target=check_passwords, args=(corpus_chunk, first_salt, pa
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

print("Password checking complete.")
```

We were able to multi-thread the password cracking, utilizing all 8 cores on

User	Password	Time Taken (seconds)
Bilbo	welcome	14.18
Gandalf	wizard	27.20
Thorin	diamond	221.41
Fili	desire	125.50
Kili	ossify	563.33
Balin	hangout	54.06
Dwalin	drossy	120.57
Oin	ispaghul	455.94
Gloin	oversave	2125.18
Dori	indoxylic	964.47
Nori	swagsman	2150.00
Ori	airway	1086.86
Bifur	corrosible	1711.43
Bofur	libellate	3202.01
Durin	purrone	3362.33

Table 2: Password Cracking Results

an M1 Macbook Air, which significantly sped up our results. Additionally, we seemed to have been lucky when cracking the work factor 13 password, as its time is only slightly longer than the longest work factor 12.

Question 1: 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?

Assuming a work factor of 12, we were able to crack a password of 1 word in an average of 2000 seconds. To crack a password of form word1:word2, we would need to try every first word, with every second word, which would result in a search space of $135000 * 135000$, which assuming results similar to before would result in a time of $2000 * 2000 = 4000000$ seconds, or 46 days.

Question 2: What about word1:word2:word3?

Using similar logic as above, this would take $2000*2000*2000 = 8*10^9$ seconds, or 253 years.

Question 3: What about word1:word2:number where number is between 1 and 5 digits?

There are 99999 possible numbers between 1 and 5 digits, thus our search space becomes $13500 * 13500 * 99999$, which assuming similar times as before, would result in $2000 * 2000 * (2000 * \frac{99999}{135000}) = 5.92 * 10^9$ seconds, or 188 years.