Assignment Group 9

Arian Houshmand
Lucas Summers

Module 4 Assignment - Hashing and Passwords

Task 1:

```
def sha256_hash(data):
   return hashlib.sha256(data.encode()).hexdigest()

def bit_flip(string, bit_pos):
   byte_list = bytearray(string.encode())
   byte_list[bit_pos // 8] ^= 1 << (bit_pos % 8)   # Flip the bit
   return byte_list.decode(errors='ignore')

def truncate_hash(hex_digest, bits):
   binary_digest = bin(int(hex_digest, 16))[2:].zfill(256)
   return binary_digest[:bits]
```

Simple helper functions to create a SHA256 hash, truncating hashes, as well as flipping a single bit (needed for part B).

Part A.)

```
# Part (a): SHA-256 Hashing
test_strings = ["hello", "world", "test123", "crypto"]
for s in test_strings:
    print(f"SHA-256({s}) = {sha256_hash(s)}")
```

Tests hashing different strings using SHA256

Output:

```
SHA-256(hello) = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
SHA-256(world) = 486ea46224d1bb4fb680f34f7c9ad96a8f24ec88be73ea8e5a6c65260e9cb8a7
SHA-256(test123) =
ecd71870d1963316a97e3ac3408c9835ad8cf0f3c1bc703527c30265534f75ae
SHA-256(crypto) = da2f073e06f78938166f247273729dfe465bf7e46105c13ce7cc651047bf0ca4
```

Part B.)

```
# Part (b): Hashing two strings with a 1-bit difference
   base_string = "hello"
   for i in range(3):
       bit_flipped_string = bit_flip(base_string, i)
       print(f"Original: {base_string}, Hash: {sha256_hash(base_string)}")
       print(f"Modified: {bit_flipped_string}, Hash:
{sha256_hash(bit_flipped_string)}")
       print("-" * 50)
```

This changes the 1st, 2nd, and 3rd bit of the base_string, which just changes the first character, then prints the resulting 256 bit hashes.

Output:

```
Original: hello, Hash:
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
Modified: iello, Hash:
db365f3eb197faeb1fc6a4038f04bdc5c0bf4185e63bc72eb50013bc122c5377
--------------------------------------------------
Original: hello, Hash:
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
Modified: jello, Hash:
187c9bceeb919e1b3e6d20fa50ecabf7d9d50b5343e8f9a3d912abb13929102e
--------------------------------------------------
Original: hello, Hash:
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
Modified: lello, Hash:
efb478ecda2d393d8e866ebce3c937bf136940d2c244242c341b08c3a0fd69cc
--------------------------------------------------
```

As the output shows, even though only 1 bit changes in the input, the resulting hashes are completely different with no visible pattern due to the unpredictable nature of the SHA256 hash function.

Part C.)

```
# Part (c): Finding Collisions
def find_collision(bits):
   start_time = time.time()
   hash_dict = {}
   attempts = 0

   while True:
       random_input = str(random.getrandbits(256))
       hex_digest = sha256_hash(random_input)
       truncated_hash = truncate_hash(hex_digest, bits)

       if truncated_hash in hash_dict:
           elapsed_time = time.time() - start_time
           return attempts, elapsed_time, random_input,
hash_dict[truncated_hash]
       hash_dict[truncated_hash] = random_input

       attempts += 1
```
This function loops until it finds a collision between two hashes of size ‘bits’ bits. Uses the birthday method by storing the seen hashes in a python dictionary and checking if new hashes are already in the dictionary.

```
    bit_sizes = list(range(8, 51, 2))  # 8-bit to 50-bit truncation in
steps of 2
    num_inputs_list = []
    time_list = []

    for bits in bit_sizes:
        attempts, elapsed_time, input1, input2 = find_collision(bits)
        print(f"Collision found at {bits} bits: {input1} and {input2}")
        num_inputs_list.append(attempts)
        time_list.append(elapsed_time)

    # Plot graphs
    plt.figure(figsize=(10, 5))

    plt.subplot(1, 2, 1)
    plt.plot(bit_sizes, time_list, marker='o', linestyle='-')
    plt.xlabel("Digest Size (bits)")
    plt.ylabel("Time to Find Collision (seconds)")
    plt.title("Digest Size vs Collision Time")
```

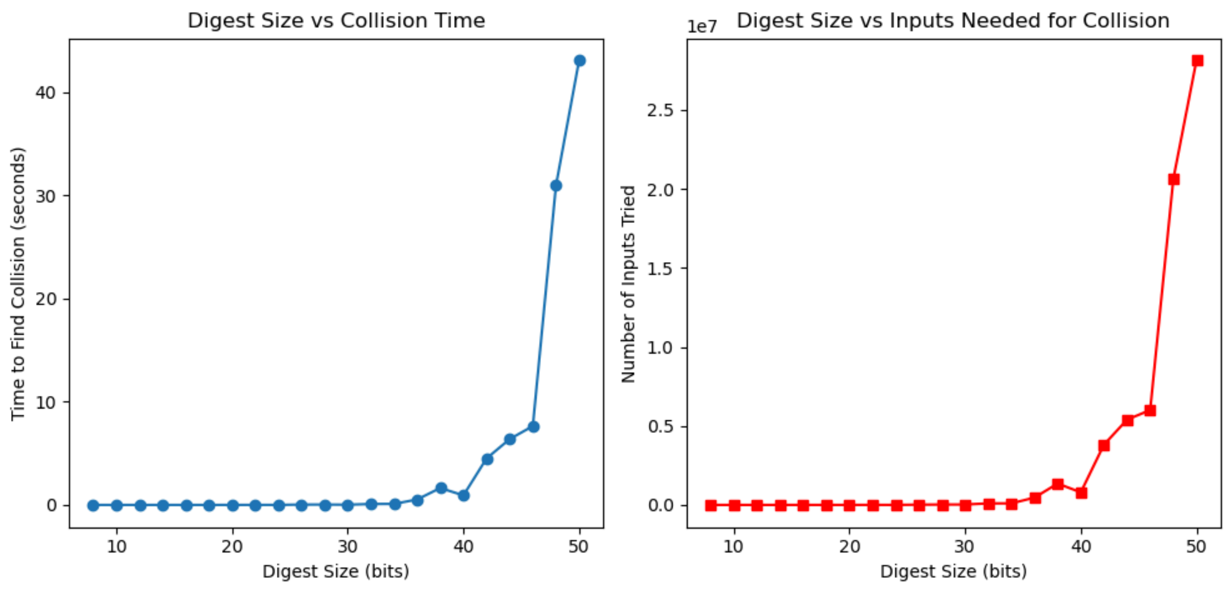
```
   plt.subplot(1, 2, 2)
   plt.plot(bit_sizes, num_inputs_list, marker='s', linestyle='-',
color='r')
   plt.xlabel("Digest Size (bits)")
   plt.ylabel("Number of Inputs Tried")
   plt.title("Digest Size vs Inputs Needed for Collision")

   plt.tight_layout()
   plt.show()
```
Finds collisions for 8 through 50 bit hashes using find_collision function above and outputs results. Also, we use matplotlib to automatically generate the graphs from the runtime and number of attempts data.

Output:
```
Collision found at 8 bits:
57780746080846506323518034521232169273676048165516065852684750774069206632932
and
11399153556577644383741439654163763700936639862524251611204068012556002695620
Collision found at 10 bits:
21627528549432210842464942893517187589400952074890532895241508112894416242942
and
14383578074172226813078102572636653935161437802605018745079075696433634428998
Collision found at 12 bits:
82688039426869083720634700619597494036094336196011008602722680576952654359888
and
61213805309149976940319437594315932982019559050258341304119213970162610251195
Collision found at 14 bits:
61166036800268406881262968385723510580986516487015974901742275899170256438954
and
90017178244266608552228180269023514524275489855895909187124588258141686621636
Collision found at 16 bits:
72938278105234069025718592424100418358542352783543087868747846614223119468621
and
112518428622676800267097128089374980076156500867196814887256300698893393873818
Collision found at 18 bits:
91266323830552866436714002531877718517490112497628083058435545991977584361910
and
75968633643478099335679132911771070303524537021151209987036623271261312152524
Collision found at 20 bits:
29878204284729363706272220489403726349314760315635176258864179844938377715660
```

```
and
68394335631112278990265706320528575674827342706378817179872031986887080510051
Collision found at 22 bits:
107383889648898438090791448933555199417383024117827945454171608630003503281238
and
67734255642973183150931069342565629927344509454407817060135864211155809538449
Collision found at 24 bits:
61651361077381992990438737725833298537996639408972275621349748604433621934266
and
15389047481199628020622711326400390310252560226925485472596193751046031480042
Collision found at 26 bits:
112916229719388385252562303537537268572425942973873699246266514960276725024768
and
60139556520168957581198799712250184548401386426029051624870135689564643456580
Collision found at 28 bits:
32908383617988120399310350404694909749438025554135916517218003117118500364885
and
106981253857069375265391438421813152827416719293283009309799203442001991531830
Collision found at 30 bits:
48380872522526047378181577793329302846003389929622830351978208199640285275938
and
31623765665434119395164750341677739643755159270801107506472808243475150159684
Collision found at 32 bits:
48175866227609081119771583441758547351353452648458140510173686548119157457745
and
6626024586443972416586672733975530935116609301691824952165134880478240272891
Collision found at 34 bits:
88561364472118956443637270698017292263376261251556567282364095870053669282094
and
74567846032874966925519004480153909701334922979124998839811937112605897316721
Collision found at 36 bits:
84902816534626032507073102657183763773055126819599832721647843535575764369172
and
86394849139574839999037866281577463305536117889225317772042863936353628270194
Collision found at 38 bits:
58018277454602863379140631774240636732465203059646964304610733812164345555165
and
65382422194799451052218563098390847642245115401066634548992679172866455076986
Collision found at 40 bits:
17745589260611258633116851457750383298791303308290703391359853470630299418105
```

and
3873153959744776836264587491588233737549106118604395486370704381284021276685 7
Collision found at 42 bits:
29114553630975137875901794510360711050185299777402478218780316023484495261242
and
74675386341995382718930718290631522301952860978746310359537271463577222744483
Collision found at 44 bits:
87986271350517821138042012422974387267356693848707249101353846196275035072131
and
7170033191081236581205274145432025111692529906435469311983987514181955419091
Collision found at 46 bits:
65430753717925129468352734253974287416144755616034664024236666192661280807471
and
11434482043260797850572720537249109156305744260732367979475192766162060845758
Collision found at 48 bits:
12195228276781870296118492572012444236707638344055594828302677064247576205374
and
98629535057052668783324795650012692460530380827185656606411418564181195778974
Collision found at 50 bits:
3650029211059312174692773675035454507435768045226549858800727364386596097728
and
6530747924864148670791050947882930144798748533084299437923275781776719535761



As the graph reveals, time to find a collision and number of inputs tried follow an almost identical trend obviously because it takes more time to try more inputs for a collision. At around a 36 bit digest size, the graphs start to trend up, with it shooting up at a fast rate around 46 bit digest size. We would assume a digest size greater than 50 would continue to follow this exponential trend, and soon would become too difficult to calculate by an average computer in a reasonable amount of time.

Task 2:

```
nltk.download('words')
wordlist = [w.lower() for w in words.words() if 6 <= len(w) <= 10]
```

Loads the dictionary wordlist from NTLK corpus, only taking words between 6 and 10 characters in length because that's all we need for the task.

```
def parse_shadow_file(filename):
   user_data = {}
   with open(filename, "r") as f:
       for line in f:
           parts = line.strip().split("$")
           if len(parts) < 4:
               continue
           user = parts[0].split(":")[0]
           salt = f"${parts[1]}${parts[2]}${parts[3][:22]}"
           hashed = f"${parts[1]}${parts[2]}${parts[3]}"
           user_data[user] = (salt, hashed)
   return user_data
```

Parses lines of the shadow file, extracting the salt and hash value for every user.

```
def check_password(candidate, hashed_password):
     return bcrypt.checkpw(candidate.encode(), hashed_password.encode())
```

Uses bcrypt to check the word against the bcrypt hash, returning true if there is a match.

```
def crack_user_password(user, salt, hashed_password):
   start_time = time.time()

   with concurrent.futures.ThreadPoolExecutor(max_workers=8) as executor:
       future_to_word = {executor.submit(check_password, word,
hashed_password): word for word in wordlist}

       with tqdm(total=len(wordlist), desc=f"Cracking {user}", unit="
tries") as pbar:
           for future in concurrent.futures.as_completed(future_to_word):
               word = future_to_word[future]
```

```
                if future.result():
                    end_time = time.time()
                    print(f"\nPassword for {user}: {word} (Time: {end_time
- start_time:.2f}s)")
                    return user, word, end_time - start_time
                pbar.update(1)

    return user, None, None
```

Uses multithreading to parallelize checking password hashes against dictionary words. We also implement a progress bar to see how many words from the wordlist have been checked so far.

```
def parallel_brute_force_bcrypt(user_data):
    cracked_passwords = {}

    with concurrent.futures.ThreadPoolExecutor(max_workers=4) as executor:
        futures = {executor.submit(crack_user_password, user, salt,
hashed): user for user, (salt, hashed) in user_data.items()}

        for future in concurrent.futures.as_completed(futures):
            user, password, time_taken = future.result()
            if password:
                cracked_passwords[user] = (password, time_taken)

    return cracked_passwords
```

We also use multithreading here to check passwords for multiple users at the same time and store the cracked password and time taken to crack it for each user.

```
if __name__ == "__main__":
    shadow_file = "shadow.txt"
    user_data = parse_shadow_file(shadow_file)
    cracked_passwords = parallel_brute_force_bcrypt(user_data)

    with open("cracked_results.txt", "w") as f:
        for user, (password, time_taken) in cracked_passwords.items():
            f.write(f"{user}: {password}, Time: {time_taken:.2f}s\n")
```

Loads the provided shadow file and runs the brute force attack on the password hashes. We store the results in an output file to refer to later.

Output:

```
Bilbo: welcome, Time: 635.35s
Gandalf: wizard, Time: 642.93s
Thorin: diamond, Time: 150.18s
Fili: desire, Time: 285.16s
Kili: ossify, Time: 805.53s
Dwalin: drossy, Time: 705.21s
Balin: hangout, Time: 1020.84s
Oin: ispaghul, Time: 1190.41s
Gloin: oversave, Time: 3061.71s
Nori: swagsman, Time: 4092.79s
Dori: indoxylic, Time: 2060.71s
Ori: airway, Time: 202.88s
Bifur: corrosible, Time: 1905.55s
Bofur: libellate, Time: 4629.34s
Durin: purrone, Time: 11229.41s
```

Questions:

1. **What do you observe based on Task 1b? How many bytes are different between the two digests?**
   Changing the input by just one bit fully changes the resulting hash in a seemingly random way with no patterns. This makes almost all the bytes different between the two digests. This is due to the "avalanche effect" of SHA256, which ensures even a slight change propagates across the entire digest, making it extremely resistant to collision attacks.

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 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.**

   The maximum number of inputs to find a collision on an n-bit digest would be 2^n, which is the total number of possible unique hashes. Due to probabilities and the birthday bound, the expected number of hashes to find a collision is only about $\sqrt{2^n} = 2^{n/2}$. This is exactly what we observed, as both the time and number of inputs tried grow exponentially as digest size grows, with a sharp increase at around 40 bits. This aligns well with the birthday bound prediction.

For a 256 bit digest, the expected number of hashes to find a collision would be:
$2^{256/2} = 2^{128} \approx 3.4 \times 10^{38}$
If the system hashes 1 billion hashes per second, the expected time would be:
$\frac{2^{128}}{10^{9}} = 1.08 \times 10^{22} sec \approx 3.4 \times 10^{14}\ years$ (longer the age of the universe!)

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?**

   For an 8 bit digest there are only $2^{8} = 256$ possible hash values, and thus to find any pre-image you could simply brute force all 256 possible inputs, which is trivial. Finding a collision using the birthday attack is easier than breaking the one-way property due to the birthday bound, which makes the expected number of hashes needed only $2^{8/2} = 16$ instead of 256.

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**

   Given our results above, the average time to brute force a password with the NLTK corpus is around 2000 seconds (or 33.33 minutes).

   word1:word2: Given there are 135,145 words (between 6 and 10 chars) in the wordlist, there are $135145^{2} = 1.83 \times 10^{10}$ two word combinations. Thus, given the average time to brute force a single word being around 2000 seconds, the time to crack is around:
   $(1.83 \times 10^{10}) \times 2000 = 3.65 \times 10^{13} sec = 1158306.13\ years$

   word1:word2:word3: There are $135145^{3} = 2.47 \times 10^{15}$ three word combinations. Thus, the time to crack is around:
   $(2.47 \times 10^{15}) \times 2000 = 4.94 \times 10^{18} sec = 156.54\ billion\ years$

   word1: words2:number: There are $135145^{2} \times 10^{5} = 1.83 \times 10^{15}$ two word and 1 number combinations. Thus, the time to crack is around:
   $(1.83 \times 10^{15}) \times 2000 = 3.65 \times 10^{18} sec = 115.83\ billion\ years$