CS4097 Assessment 1 Report

Ben Temming

Contents

1. Task 1: Brute-force Cracking

The bruteforce_hashes() function starts by initializing a result list and a set tracking uncracked hashes. Next, the function sets the initial password length to 1 so that password guesses can be checked in shortlex order. The function's core operates within a while loop, executing as long as there are uncracked hashes. At each iteration, a list of all possible permutations of the password character for the current password length is generated. For each permutation, the SHA512 hash is computed and compared to the list of provided hashes. If a match is found, the password is stored in the result list at the same index position as the matching hash in the input hash list, ensuring that the passwords are associated with their respective hashes. After processing all permutations, the function increments the password length and repeats the process until all hashes have been cracked. A "break" statement is triggered to exit the loop when all hashes have been cracked. This greatly improves performance as it prevents unnecessary iteration of permutations. Finally, the function returns a list containing the discovered passwords.

Passwords corresponding to the provided hashes: ['m', 'mc', '555', 'admi']

Refer to Appendix A for the code.

2. Task 2: Dictionary Cracking

Dictionary cracking, implemented in the dictionary_crack_hashes() function, involves comparing known password hashes with the hashes that should be cracked. In this task, a dictionary file "PasswordDictionary.txt" obtained from GitHubGist[1] is used for this purpose. The dictionary_crack_hashes() function works in a similar manner to the bruteforce_hashes() function from task1, but unlike brute force, which exhaustively tests character combinations, the dictionary approach iterates through each word in the dictionary, computes its hash, and performs hash matches. The dictionary is a list of commonly used passwords, so using a dictionary word instead of a randomly generated sequence of characters has a much higher chance of being the right password. This method significantly improves the efficiency of cracking a hash, as using a frequently used password is akin to making an educated guess and reduces the number of hashes that have to be computed significantly.

Passwords corresponding to the provided hashes: ['45678', 'admin$', 'Blessing', 'Windows10', 'Unicorn@1234', 'Qwertyuiop2016', 'Passw0rd!@#', '~!@#$%^&*()_+', 'Zxc123!@#', 'Behappy']

Refer to Appendix B for the code.

[1] Staev

3. Task 3: Dictionary Cracking with Salts

Dictionary cracking with salts, as implemented in the dictionary_crack_salted_hash() function, is an extension of dictionary cracking. It involves appending a unique salt to each password guess, hashing the combination, and comparing it to the password hashes to be cracked. Similar to task 2, the “PasswordDictionary.txt” dictionary is utilised, but with a distinct salt for each hash, the function can only crack one hash at a time. This means it has to iterate over each hash-slat pair and apply dictionary cracking where the salt is added to each dictionary word. The inclusion of a salt prevents precomputing dictionary word hashes and comparing them to multiple target hashes. Consequently, the algorithm's performance is notably affected, as it requires hashing the full dictionary for each hash-salt pair to successfully crack each hash in the provided list of hashes.

Passwords corresponding to the provided hashes: ['MTC123', 'Hetzneronline!@1234', 'Server!@#$', 'Welcome!', 'Admin121!@#$%^', 'Zxcv123$', 'G00dluck', 'Vps@@##11', 'Winner!@#', 'Lovemyself']

Refer to Appendix C for the code.

4. Task 4: Password-based key derivation

4.1. Goal of Password-based key derivation

As stated in the National Institute of Standards and Technology [NIST] Special Publication 800-132, user-chosen passwords are not suitable to be used directly as cryptographic keys as they often have low entropy and poor randomness[2]. Therefore, it is essential to use a password-based key derivation function [PBKDF] that can transform the cryptographically insecure password into a usable cryptographic key which can be used to protect data. A PBKDF protects against brute force and dictionary attacks by increasing the computational cost of calculating the hash of a password. This increased cost has a minimal effect on the users, as the password only has to be hashed once, but greatly affects hash-cracking algorithms as it has to hash millions of password guesses. Furthermore, PBKDFs also utilize salts that further increase the complexity of cracking a list of password hashes that could have been stolen in an attack. The goal for this task is to implement PBKDF2, which is currently recommended by NIST[3], and compare its efficiency to the SHA512 hashing algorithm.

4.2. Implementation of PBKDF2

The pseudo-code for the PBKDF2 algorithm is given on page 7 of the NIST Special Publication 800-132[4] and can be summarized as follows. Given a password, P, a salt, S, an iteration count, C, and the length of the master key kLen, use a keyed hashing for message authentication [HMAC] function to generate a master key of length kLen. Depending on the digest size of the hashing algorithm used in the HMAC, in this case SHA512, the master key has to be generated by generating blocks and then concatenating these blocks to create the master key whose length matches kLen. To generate a block, the HMAC is applied C times, each time with the password and a message that is initialized using the salt and updated to the output of the HMAC at each iteration. Updating the message with the HMAC output at each iteration ensures that an attacker cannot simply skip the iterations by using some other function and has to iterate the specified number of times, which makes a brute force or dictionary attack very inefficient. After each iteration, the updated HMAC output is XORed with the block value initially set to 0 and updated at each iteration with the result of the XOR operation. At the

[2] Sönmez p. 1
[3] Password Storage Cheat Sheet
[4] Sönmez

end of the algorithm the master key, which corresponds to all generated blocks concatenated in order of generation, is returned.

The implementation of PBKDF2 differs slightly from the NIST pseudo-code but the fundamental algorithm is the same. For instance, the implemented PBKDF2() function takes the password and the salt input as a string and assumes that the length of the desired master key is provided in bytes rather than bits. The function also returns the final key in hexadecimal instead of binary, as this makes the output more readable and allows direct comparison to the output of the hash_str() function implemented in task 3. The implementation utilises the hmac library[5] to calculate the HMAC value used in the XOR operation. This is done because NIST states that their recommendation "approves PBKDF2 as the PBKDF using HMAC with any approved hash function as the PRF"[6]. SHA512 is selected as the PRF for the HMAC as this allows for a better performance comparison of PBKDF2 and SHA512

Refer to Appendix D for the code that implements PBKDF2.

4.3. PBKDF2 vs SHA512 Performance

To compare the performance of PBKDF2 to SHA512 the timeit library[7] can be used to measure the execution time of both functions. To get a more accurate comparison between SHA512 and PBKDF2, the same password and salt are used for both, and the length of the desired master key is set to 64 bytes, as this corresponds to the 512-bit hash generated by SHA512. The result can be seen below in Figure 1.

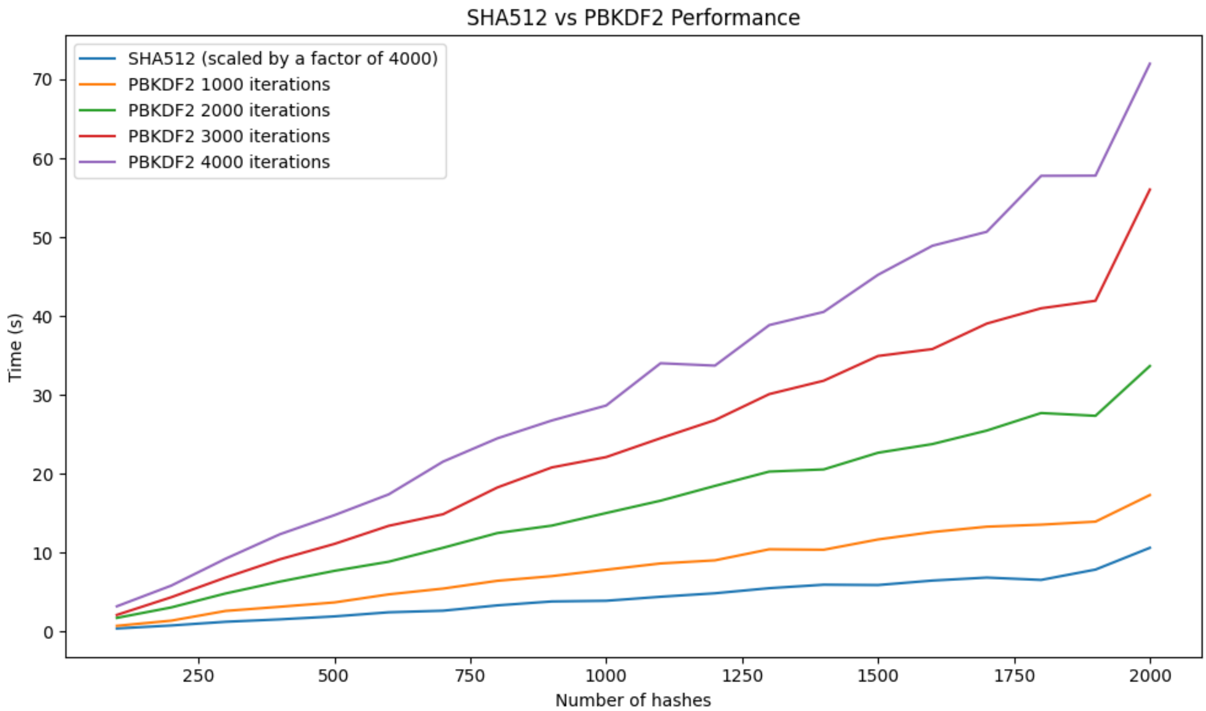

Figure 1: Execution time of SHA512 and PBKDF2 for a specified number of hashes.

[5] hmac
[6] Sönmez p. 7
[7] timeit

As expected, Figure 1 shows that PBKDF2's execution time increases with an increased number of iterations and that SHA512 is significantly faster than PBKDF2. It is important to note that the values for SHA512 had to be scaled by a factor of 4000 to appear non-zero when placed on the same scale as PBKDF2. A factor of 4000 was chosen because it highlights the fact that a single iteration in PBKDF2 is more computationally expensive than calculating a SHA512 hash, otherwise, the line of SHA512 execution time scaled by a factor of 4000 would show a similar performance as PBKDF2 with 4000 iterations.

Even though applying PBKDF2 with 4000 iterations is already a significant increase in computational cost compared to SHA512, the Open Web Application Security Project [OWASP] currently recommends 210,000 iterations for PBKDF2 with a SHA512 HMAC function[8]. This is around 50 times more iterations than the highest tested iteration count and makes it evident that brute forcing or using a dictionary attack for any password encoded with PBKDF2 is extremely inefficient.

4.4. Conclusion

Even though PBKDF2 is a great improvement compared to simply using SHA512 with a salt to encrypt passwords, PBKDF2 still has some shortcomings. For instance, it has been found that "PBKDF2 can be often implemented very efficiently on GPUs, thereby providing an attacker a huge potential speedup compared to the defender (who almost always runs PBKDF2 on a CPU)"[9]. Thus, with ever-better hardware the encryption algorithms must be adapted, and new methods have to be implemented that are even harder to crack.

Overall, I have learned that even though most areas of computer science focus on creating algorithms that are as efficient as possible, security purposely creates algorithms that are inefficient and take large amounts of memory to protect against bad actors. Furthermore, the world of computer security is a cat-and-mouse game where researchers have to constantly create new, more secure methods before a bad actor manages to compromise secrets using new techniques or stronger hardware.

---

[8] Password Storage Cheat Sheet

[9] Visconti p. 1

5. References

hmac - Keyed-Hashing for Message Authentication, Python 3.12.0 documentation, Python Software Foundation, https://docs.python.org/3/library/hmac.html (Accessed 14.10.2023)

Password Storage Cheat Sheet, OWASP Cheat Sheet Series, Available at https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#pbkdf2 (Accessed 12.10.2023)

Staev, Peter, Password dictionary, Github, https://gist.github.com/PeterStaev/e707c22307537faeca7bb0893fdc18b7 (Accessed 1.10.2023)

Sönmez Turan, Meltem; Barker, Elaine; Burr, William; Chen, Lily. "Recommendation for Password-Based Key Derivation Part 1: Storage Applications" (PDF). NIST. SP 800-132.

Available at https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf (Accessed 12.10.2023)

timeit – Measure execution time of small code snippets, Python 3.12.0 documentation, Python Software Foundation, https://docs.python.org/3/library/timeit.html (Accessed 14.10.2023)

Visconti, Andrea; Mosnácek, Ondrej; Brož, Milan; Matyáš, Vashek. “Examining PBKDF2 security margin – Case study of Luks” (PDF). Journal of Information Security and Applications 46 (2019) 296-306. Available at https://www.sciencedirect.com/science/article/pii/S221421261730025X?ref=pdf_download&fr=RR-2&rr=8168eed67e6ddd60 (Accessed 12.10.2023)

6. Appendix

6.1. Appendix A: Task 1 Code

```
import itertools
import hashlib


def hash_tuple(password_tuple):
    #create string from tuple
    password_str = "".join(password_tuple)

    ##hash the password string
    #create sha512 object
    sha512 = hashlib.sha512()
    #update object with encoded password_str
    sha512.update(password_str.encode())
    #generate and return hashed string
    return password_str, sha512.hexdigest()


def get_all_permuations(elem_list, length):
    #we want to get all possible permuations with repetition and a set length
    return list(itertools.product(elem_list, repeat=length))


def bruteforce_hashes(hash_list):
    password_chars = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m",
                      "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z",
                      "1", "2", "3", "4", "5", "6", "7", "8", "9"]
    #store result in list to keep the order of the passwords
    result_list = ["Not found"]*len(hash_list)
    #create a set of the indexes of all the uncracked hashes
    uncracked_hashes_indexes = set(range(len(hash_list)))

    #set the initial passowrd length, shortest length is 1
    password_len = 1

    #while there are still uncracked passwords
    while uncracked_hashes_indexes:
        #get all possilbe permuations for password chars with lenght password_len
        # in lexicographical order
        permutations_list = get_all_permuations(password_chars, password_len)

        #for every permutation
        for perm in permutations_list:
            #create password string and generate hash
            perm_str, hashed_perm = hash_tuple(perm)
            #for every uncracked hash check if the there is a match
            for index in list(uncracked_hashes_indexes):
                #check if the hash from hash list is the same as the generated hash
```

```
                if hash_list[index] == hashed_perm:
                    #if a matching hash is found, update the result list with the password
                    result_list[index] = perm_str
                    #remove the hash index from the set of uncracked hashes
                    uncracked_hashes_indexes.remove(index)

            # if there are no more uncracked hashes stop checking permuations
            if not uncracked_hashes_indexes:
                break
        #increment password length by 1
        password_len +=1
    #return list of password
    return result_list


if __name__ == "__main__":
    hash_list = [
        'f14aae6a0e050b74e4b7b9a5b2ef1a60cec-
cbbca39b132ae3e8bf88d3a946c6d8687f3266fd2b626419d8b67dcf1d8d7c0fe72d4919d9bd05efbd37070cfb4
1a',
        'e85e639da67767984cebd6347092df661ed79e1ad21e402f8e7de01fdedb5b0f165cbb30a20948f1ba
3f94fe33de5d5377e7f6c7bb47d017e6dab6a217d6cc24',
        '4e2589ee5a155a86ac912a5d34755f0e3a7d1f595914373da638c20fecd7256ea1647069a2bb48ac42
1111a875d7f4294c7236292590302497f84f19e7227d80',
        'afd66cdf7114eae7bd91da3ae49b73b866299ae545a44677d72e09692cdee3b79a022d8dcec9994835
9e5f8b01b161cd6cfc7bd966c5becf1dff6abd21634f4b'
        ]
    #get list of passwords
    password_list = bruteforce_hashes(hash_list)

    #print each password in the terminal
    for password in password_list:
        print(password)
```

6.2. Appendix B: Task 2 Code

```
import hashlib

def hash_str(password_str):
    #create sha512 object
    sha512 = hashlib.sha512()
    #update object with encoded password_str
    sha512.update(password_str.encode())
    #generate and return hashed string
    return sha512.hexdigest()


def dictionary_crack_hashes(dict_vals, hash_list):
    result_list = ["Not found"]*len(hash_list)
    uncracked_hashes_indexes = set(range(len(hash_list)))
```

```
    dict_counter = 0 #keeping track of which password to check in the dict_vals list
    dict_len = len(dict_vals)

    while uncracked_hashes_indexes and (dict_counter < dict_len):
        #select the dictionary password to check
        password_str = dict_vals[dict_counter]
        #compute hash of the password
        password_hash = hash_str(password_str)

        #for every uncracked hash check if there is a match
        for index in list(uncracked_hashes_indexes):
            #check if the hash from hash list is the same as the password hash
            if hash_list[index] == password_hash:
                #matching hash, update result list with password and remove the hash
                # index from uncracked hashes set
                result_list[index] = password_str
                uncracked_hashes_indexes.remove(index)

        # if there are no more uncracked hashes stop checking dictionary
        if not uncracked_hashes_indexes:
            break
        #move to next word in dictionary list
        dict_counter += 1

    return result_list


if __name__ == "__main__":

    #load dictionary values
    file_name = "PasswordDictionary.txt"
    password_dictionary_list = []
    with open(file_name, "r") as file:
        #read each line in the file, strip spaces and add password to list
        for line in file:
            password_dictionary_list.append(line.strip())

    #list of hashes that we want to crack
    hash_list = [
        '31a3423d8f8d93b92baffd753608697ebb695e4fca4610ad7e08d3d0eb7f69d75cb16d61caf7cead05
46b9be4e4346c56758e94fc5efe8b437c44ad460628c70',
        '9381163828feb9072d232e02a1ee684a141fa9cddcf81c619e16f1dbbf6818c2edcc7ce2dc053eec39
18f05d0946dd5386cbd50f790876449ae589c5b5f82762',
        'a02f6423e725206b0ece283a6d59c85e71c4c5a9788351a24b1ebb18dcd8021ab854409130a3ac941f
a35d1334672e36ed312a43462f4c91ca2822dd5762bd2b',
        '834bd9315cb4711f052a5cc25641e947fc2b3ee94c89d90ed37da2d92b0ae0a33f8f7479c2a57a32fe
abdde1853e10c2573b673552d25b26943aefc3a0d05699',
```

```
        '0ae72941b22a8733ca300161619ba9f8314ccf85f4bad1df0dc488fdd15d220b2dba3154dc8c78c577
979abd514bf7949ddfece61d37614fbae7819710cae7ab',
        '6768082bcb1ad00f831b4f0653c7e70d9cbc0f60df9f7d16a5f2da0886b3ce92b4cc458fbf03fea094
e663cb397a76622de41305debbbb203dbcedff23a10d8a',
        '0f17b11e84964b8df96c36e8aaa68bfa5655d3adf3bf7b4dc162a6aa0f7514f32903b3ceb53d223e74
946052c233c466fc0f2cc18c8bf08aa5d0139f58157350',
        'cf4f5338c0f2ccd3b7728d205bc52f0e2f607388ba361839bd6894c6fb8e267beb5b5bfe13b6e8cc5a
b04c58b5619968615265141cc6a8a9cd5fd8cc48d837ec',
        '1830a3dfe79e29d30441f8d736e2be7dbc3aa912f11abbffb91810efeef1f60426c31b6d666eadd83b
bba2cc650d8f9a6393310b84e2ef02efa9fe161bf8f41d',
        '3b46175f10fdb54c7941eca89cc813ddd8feb611ed3b331093a3948e3ab0c3b141ff6a7920f9a068ab
0bf02d7ddaf2a52ef62d8fb3a6719cf25ec6f0061da791'
        ]

    #get list of passwords
    password_list = dictionary_crack_hashes(password_dictionary_list, hash_list)
    #print each password in the terminal
    for password in password_list:
        print(password)
```

6.3. Appendix C: Task 3 Code

```
import hashlib

def hash_str(password_str):
    #create sha512 object
    sha512 = hashlib.sha512()
    #update object with encoded password_str
    sha512.update(password_str.encode())
    #generate and return hashed string
    return sha512.hexdigest()


def dictionary_crack_salted_hash(dict_vals, password_hash, salt):
    #loop over every password dictionary value
    for dict_val in dict_vals:
        #create string with dict val and salt
        salted_dict_val = dict_val + salt
        #get hash of salted value
        hashed_val = hash_str(salted_dict_val)
        #check if hash matches password hash
        if hashed_val == password_hash:
            #if match is found, return dict val
            return dict_val

    #if no match is found, return default value
    return "Password Not Found"


def dictionary_crack_salted_hashes(dict_vals, hash_salt_list):
```

```
    result_list = []
    #for every salted hash find the password string
    for hash_salt in hash_salt_list:
        password_str = dictionary_crack_salted_hash(dict_vals, hash_salt[0], hash_salt[1])
        result_list.append(password_str)

    return result_list


if __name__ == "__main__":
    #load dictionary values
    file_name = "PasswordDictionary.txt"
    password_dictionary_list = []
    with open(file_name, "r") as file:
        #read each line in the file, strip spaces and add password to list
        for line in file:
            password_dictionary_list.append(line.strip())

    #list of hashes that we want to crack
    hash_salt_list =  [
        ('63328352350c9bd9611497d97fef965bda1d94ca15cc47d5053e164f4066f546828eee451cb5edd6f
2bba1ea0a82278d0aa76c7003c79082d3a31b8c9bc1f58b',
         'dbc3ab99'),
        ('86ed9024514f1e475378f395556d4d1c2bdb681617157e1d4c7d18fb1b992d0921684263d03dc4506
783649ea49bc3c9c7acf020939f1b0daf44adbea6072be6',
         'fa46510a'),
        ('16ac21a470fb5164b69fc9e4c5482e447f04f67227102107ff778ed76577b560f62a586a159ce8267
80e7749eadd083876b89de3506a95f51521774fff91497e',
         '9e8dc114'),
        ('13ef55f6fdfc540bdedcfafb41d9fe5038a6c52736e5b421ea6caf47ba03025e8d4f83573147bc06f
769f8aeba0abd0053ca2348ee2924ffa769e393afb7f8b5',
         'c202aebb'),
        ('9602a9e9531bfb9e386c1565ee733a312bda7fd52b8acd0e51e2a0a13cce0f43551dfb3fe2fc5464d
436491a832a23136c48f80b3ea00b7bfb29fedad86fc37a',
         'd831c568'),
        ('799ed233b218c9073e8aa57f3dad50fbf2156b77436f9dd341615e128bb2cb31f2d4c0f7f8367d7cd
eacc7f6e46bd53be9f7773204127e14020854d2a63c6c18',
         '86d01e25'),
        ('7586ee7271f8ac620af8c00b60f2f4175529ce355d8f51b270128e8ad868b78af852a50174218a031
35b5fc319c20fcdc38aa96cd10c6e974f909433c3e559aa',
         'a3582e40'),
        ('8522d4954fae2a9ad9155025ebc6f2ccd97e540942379fd8f291f1a022e5fa683acd19cb8cde9bd89
1763a2837a4ceffc5e89d1a99b5c45ea458a60cb7510a73',
         '6f966981'),
        ('6f5ad32136a430850add25317336847005e72a7cfe4e90ce9d86b89d87196ff6566322d11c1367590
6883c8072a66ebe87226e2bc834ea523adbbc88d2463ab3',
         '894c88a4'),
```

```
        ('21a60bdd58abc97b1c3084ea8c89ae-
aef97d682c543ff6edd540040af20b5db228fbce66fac962bdb2b2492f40dd977a944f1c25bc8243a4061dfeeb0
2ab721e',
         '4c8f1a45')
        ]
    #get list of passwords
    password_list = dictionary_crack_salted_hashes(password_dictionary_list,
hash_salt_list)
    #print each password in the terminal
    for password in password_list:
        print(password)
```

6.4. Appendix D: Task 4 Code

```
import hashlib
import hmac
import math

#libraries used for generating performance plot
import timeit
import pandas as pd
import matplotlib.pyplot as plt
import os
#reusing the hash_str function implemented in task 3
from part3 import hash_str

#generate the hmac bytes
def HMAC(password, message):
    #calculate hmac value from password using SHA512 as the digest mode
    hmac_val = hmac.new(password, message, hashlib.sha512).digest()
    return hmac_val


def PBKDF2(password, salt, iterations, mk_len):
    #password: password of any length (string)
    #salt: randomly generated salt (string)
    #iterrations: number of itterations that should be performed,
    #  recommended bigger than (find value with sourece, use 1000 for test)
    #mk_len: the length of the master key (desired key) in bytes

    #get sha512 digest size in bytes
    sha512_digest_len = hashlib.sha512().digest_size

    #check if provided mk_len is not to large (can be at most (2**32 -1)*sha512_digest_len
bits)
    if (mk_len > (2**32 -1)*sha512_digest_len):
        raise ValueError("mk_len is too large")

    #convert password and salt to byte strings
    byte_password = password.encode()
```

```python
    byte_salt = salt.encode()

    #calculate how many blocks need to be concattinated to create key of specified size
    num_blocks = math.ceil(mk_len/sha512_digest_len)

    #initialize the master key
    byte_master_key = b""

    #generate blocks
    for i in range(1, num_blocks + 1):
        #initialize Ti = 0 as a byte string of length 64
        Ti_int = 0
        Ti = Ti_int.to_bytes(64, "big")

        #initialize U (u0) as the salt concattinated with i, where i is a 32bit integer
        # in byte format
        byte_i = i.to_bytes(4, "big")
        U = byte_salt + byte_i

        #iterate the specified number of times
        for j in range(1, iterations + 1):
            #for each iteration, the HMAC message is the previous HMAC digest, the
            # password is always the same
            U = HMAC(byte_password, U)

            #bitwise exclusive or (XOR) Ti = Ti XOR Uj
            #bytes are immutable in python so zip is needed to
            # perform the XOR operation
            Ti = bytes(Tik ^ Uk for Tik, Uk in zip(Ti, U))

        #concatinate Ti to master key to build up the key
        byte_master_key = byte_master_key + Ti

    #adjust the size of the master key to mk_len
    byte_master_key = byte_master_key[:mk_len]

    #convert to hex
    hex_naster_key = byte_master_key.hex()

    #return byte master key
    return hex_naster_key



def profile_performance(filename):
    #fixed values
    password = "test_password"
    salt = "test_salt"
    salted_password = password + salt
```

```
mk_len = 64 #64 bytes = 128 hex = 512bit, same as sha512

#varying the number of hashes that should be computed
number_of_hashes_list = [i for i in range(100, 2001, 100)]

#arrays for storing result
sha512_times = []
PBKDF2_1000_times = []
PBKDF2_2000_times = []
PBKDF2_3000_times = []
PBKDF2_4000_times = []

#generate data for different number of hashes
for number_of_hashes in number_of_hashes_list:
    print(f"Number of hashes: {number_of_hashes}")

    #time the sha512 algorithm
    sha512_time = timeit.timeit(lambda:hash_str(salted_password),
                                number=number_of_hashes*4000)
    sha512_times.append(round(sha512_time, 10))

    #time PBKDF2 alroithm with 1000 iterations
    PBKDF2_time = timeit.timeit(lambda:PBKDF2(password, salt, 1000, mk_len),
                                number=number_of_hashes)
    PBKDF2_1000_times.append(round(PBKDF2_time, 10))

    #time PBKDF2 alroithm with 2000 iterations
    PBKDF2_time = timeit.timeit(lambda:PBKDF2(password, salt, 2000, mk_len),
                                number=number_of_hashes)
    PBKDF2_2000_times.append(round(PBKDF2_time, 10))

    #time PBKDF2 alroithm with 3000 iterations
    PBKDF2_time = timeit.timeit(lambda:PBKDF2(password, salt, 3000, mk_len),
                                number=number_of_hashes)
    PBKDF2_3000_times.append(round(PBKDF2_time, 10))

    #time PBKDF2 alroithm with 4000 iterations
    PBKDF2_time = timeit.timeit(lambda:PBKDF2(password, salt, 4000, mk_len),
                                number=number_of_hashes)
    PBKDF2_4000_times.append(round(PBKDF2_time, 10))

#combine data to a panda dataframe
performance_data_dict = {
    "Number of hashes": number_of_hashes_list,
    "SHA512 time (s) (scaled by a factor of 4000)": sha512_times,
    "PBKDF2 1000 iterations time (s)": PBKDF2_1000_times,
    "PBKDF2 2000 iterations time (s)": PBKDF2_2000_times,
    "PBKDF2 3000 iterations time (s)": PBKDF2_3000_times,
    "PBKDF2 4000 iterations time (s)": PBKDF2_4000_times,
```

```
    }
    df_performance_data = pd.DataFrame(performance_data_dict)

    #save data as a CSV
    df_performance_data.to_csv(filename)


def plot_performance_data(filename):
    #load the performance data
    df_performance_data = pd.read_csv(filename, index_col=0)

    #set the figure size
    plt.figure(figsize=(10, 6))

    #dictionary for mapping the column name to legend label
    legend_dict = {
        "SHA512 time (s) (scaled by a factor of 4000)": "SHA512 (scaled by a factor of 
4000)",
        "PBKDF2 1000 iterations time (s)": "PBKDF2 1000 iterations",
        "PBKDF2 2000 iterations time (s)": "PBKDF2 2000 iterations",
        "PBKDF2 3000 iterations time (s)": "PBKDF2 3000 iterations",
        "PBKDF2 4000 iterations time (s)": "PBKDF2 4000 iterations",
    }

    #plot each column of the dataframe
    x = df_performance_data["Number of hashes"]
    columns_to_plot = [column for column in df_performance_data.columns if column != "Num-
ber of hashes"]
    for column in columns_to_plot:
        plt.plot(x, df_performance_data[column], label=legend_dict.get(column))

    #label the plot
    plt.title("SHA512 vs PBKDF2 Performance")
    plt.xlabel("Number of hashes")
    plt.ylabel("Time (s)")
    plt.legend()
    plt.tight_layout()
    #save plot as png
    plt.savefig("algorithm_performance_plot.png")
    #show the plot
    plt.show()




if __name__ == "__main__":
    #testing PBKDF2 implementation
    password = "password123"
    salt = "salt"
```

```
    iterations = 1000
    mk_len = 64 #64 bytes = 128 hexadecimal digest = 4*128 = 512 bits same as sha512

    #expected output
    test_result = 
"0ecb3c32f57685303ff0878481d223bc1a16eb13bd46cf03d275e0ed43e52104b4ca156b01abb36ee95149c8bb
bbb611b88634ffe235bd531e2a087a84d7fc85"

    #generate the master key
    master_key = PBKDF2(password, salt, iterations, mk_len)
    print(f"Generated master key: {master_key}")
    print(f"length of generated master key (bytes): {len(master_key)}")

    #check if the generated and expected key match
    if master_key == test_result:
        print("Generated master key is the same as the expected master key")


    ## generate performance plot
    #check if the performance data has already been generated
    filename = "algorithm_performance_data.csv"
    if not os.path.isfile(filename):
        #if the file does not exist, generate perfromance data
        profile_performance(filename)
    #plot performance data stored in the csv file
    plot_performance_data(filename)
```