EN4720: Security in Cyber-Physical Systems
Programming Assignment — Passwords

Name: Thalagala B. P.
Index No: 180631J

July 8, 2023

This is an individual assignment!
Due Date: 8 July 2023 by 11.59 PM

Content adopted from **CNT5410 Computer and Network Security** taught by Prof. Vincent Bindschaedler at the University of Florida.

Instructions

Please read the instructions and questions carefully. Write your answers directly in the space provided. Compile the tex document and hand in the resulting PDF as your report.

In this assignment, you will write a few lines Python code. You are encouraged to use Python3. You will need the PyCrypto library to run the provided code. Please refer to resources specific to your operating system and environment for installation instructions[1]. For example, you can use: `pip3 install pycrypto` to install PyCrypto with Python3 under Linux.

Assignment Files

The assignment archive contains the following Python source files:

- `utils.py`. This file contains utility functions needed for the assignment..
- `crypto.py`. This file defines cryptographic functions.
- `attack.py`. This file contains attack code used in the assignment.

In addition, the assignment archive contains a `data` directory which includes a dictionary file (`words.list`) and several database dump files in JSON format (`*-dbdump.json`).

Submission

Write your answers directly in the space provided. Compile the tex document and submit the resulting PDF together with the source code in a single zip file through Moodle.

Note:
You are encouraged to take a look at the provided files. This may help you successfully complete the assignment.
You might have to change the line "from Cryptodome.Hash import SHA256" in crypto.py to "from Crypto.Hash import SHA256" based on your OS and library versions.
The program can take a while to run and the terminal might not show anything until program execution is complete. But it should not take more than 5 minutes (max).

[1] https://pypi.org/project/pycrypto/ See also: https://www.pycryptodome.org/en/latest/index.html.

Problem 0: Bruteforce attack (12 pts)

In this problem, you will mount a bruteforce attack against a stolen database dump of password hashes (`data/simple-dbdump.json`). The code for this problem is already written. (But you are encouraged to take a look to see how it works.)

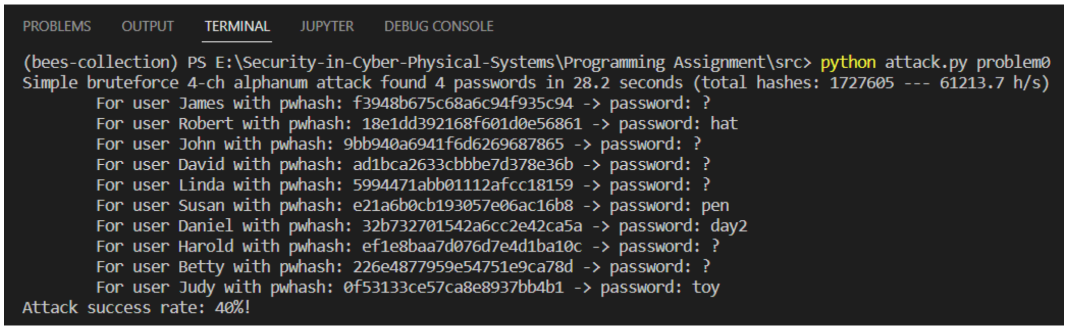
The bruteforce attack will try all possible passwords of 4 (or less) alphanumeric characters — lowercase only. To perform the attack run the following command[2]:

```
python attack.py problem0
```

1. (2 pts) What steps can you take to create a password that is hard to crack using bruteforce attacks?

   - ***A combination of numbers, letters (upper case and/or lower case) and special characters*** *must be used. This make the number of options per character to be large which eventually make it hard to brute-force.*
   - ***The password must be sufficiently long****. This makes the number of possible password combinations (password space) extremely large which will make the brute-force attack computationally infeasible to be done in a shorter period of time.*

2. (4 pts) Which passwords are recovered (for which users)? How long did the attack take (in seconds)? What was the speed of the attack (in hashes/second)? Add a screenshot of your terminal output showing the answer.



```
(bees-collection) PS E:\Security-in-Cyber-Physical-Systems\Programming Assignment\src> python attack.py problem0
Simple bruteforce 4-ch alphanum attack found 4 passwords in 28.2 seconds (total hashes: 1727605 --- 61213.7 h/s)
        For user James with pwhash: f3948b675c68a6c94f935c94 -> password: ?
        For user Robert with pwhash: 18e1dd392168f601d0e56861 -> password: hat
        For user John with pwhash: 9bb940a6941f6d6269687865 -> password: ?
        For user David with pwhash: ad1bca2633cbbbe7d378e36b -> password: ?
        For user Linda with pwhash: 5994471abb01112afcc18159 -> password: ?
        For user Susan with pwhash: e21a6b0cb193057e06ac16b8 -> password: pen
        For user Daniel with pwhash: 32b732701542a6cc2e42ca5a -> password: day2
        For user Harold with pwhash: ef1e8baa7d076d7e4d1ba10c -> password: ?
        For user Betty with pwhash: 226e4877959e54751e9ca78d -> password: ?
        For user Judy with pwhash: 0f53133ce57ca8e8937bb4b1 -> password: toy
Attack success rate: 40%!
```

Figure 1: Output of the `python attack.py problem0`

- *Time for the attack = 28.2 seconds*
- *Speed of the attack = 61213.7 hashes per second*
- *Attack success rate: 40%*

User	**Password**
Robert	hat
Susan	pen
Daniel	day2
Judy	toy

Table 1: Recovered Passwords

[2]You may have to specify to use Python3 `python3` if you also have Python2 versions installed on your system.

3. (2 pts) What is the total number of *possible* passwords of l *or less* alphanumeric characters — lowercase only?

   Since there are 26 lowercase letters and 10 digits, there are a total of 36 possible characters that can be used in each position of the password. For a password of length n, there are 36^n passwords. If we consider the passwords of length l and less, we can have the following number of total passwords, which is the sum of a geometric series where number of terms equal to l, common ratio is 36 and the first term is 36.

$$\sum_{n=1}^{l} 36^n = 36 \cdot \frac{36^l - 1}{36 - 1} = \frac{36^{l+1} - 36}{35} \tag{1}$$

   In addition, the program considers an empty string (' ') as the first candidate password. Therefore the total password count becomes,

$$\frac{36^{l+1} - 36}{35} + 1 = \frac{36^{l+1} - 1}{35}$$

4. (4 pts) Given your previous answers. How long do you estimate it would take to perform the same attack on all passwords of 8 or less alphanumeric characters — lowercase only? (Justify your answer.) Can you confirm your calculation experimentally?

   Total number of possible passwords say P,

$$P = \frac{36^{l+1} - 1}{35} = \frac{36^{8+1} - 1}{35} \approx 2.9 \times 10^{12}$$

   Average speed of the attack (this changes runtime-to-runtime and depending on the specifications of the machine used) therefore assume it to be $S = 60 \times 10^3$ hashes per second (an approximate value to the value that observed previously).

   Therefore approximate time in days say D,

$$D = \frac{2.9 \times 10^{12}}{60 \times 10^3} \times \frac{1}{24 \times 60 \times 60} \approx 560 \ days$$

   This type of attack is computationally infeasible to be launched using a personal computer. Therefore, this can not be confirmed experimentally.

Problem 1: Dictionary attack (16 pts)

In this problem, you will perform a dictionary attack against the same list of stolen password hashes (`data/simple-dbdump.json`). You will implement the (generator) function `candidate_dict_generator()` which is (partially) defined in `crypto.py`. The rest of the code is provided.

Your dictionary attack will produce password guesses of the form w **or** $w||s$ where w is a dictionary word, s is a suffix, and $||$ denotes concatenation. Refer to comments in `crypto.py` for more detailed instructions.

Run the following command to perform the attack:

```
python attack.py problem1
```

1. (3 pts) What are the differences between a dictionary attack and a bruteforce attack? What steps can you take when building passwords to defend from dictionary attacks?

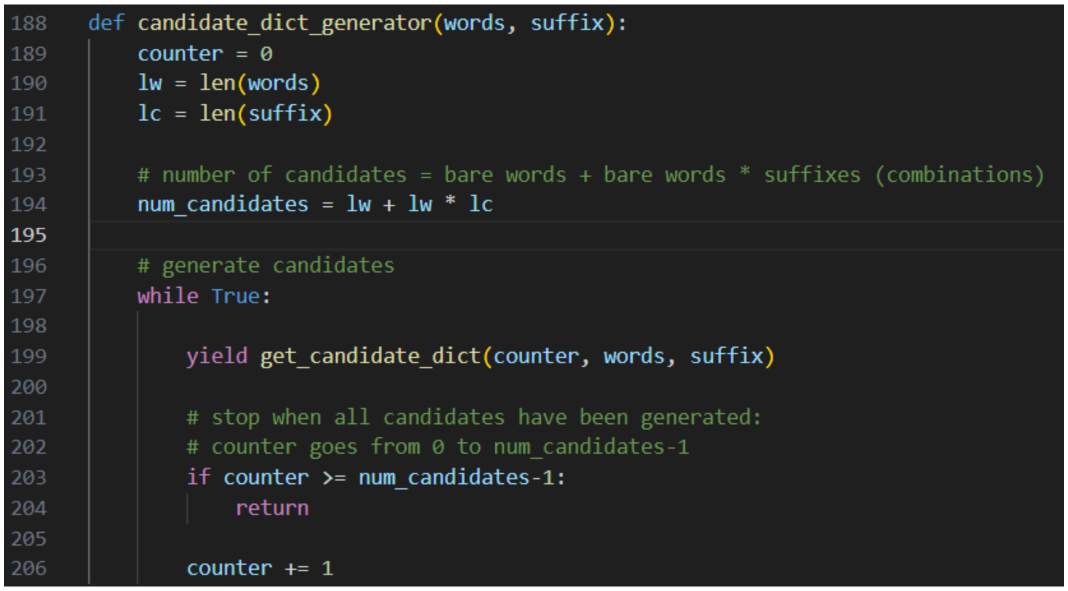
   Differences between a dictionary attack and a bruteforce attack,

   - *A brute force attack tries all possible passwords for a given length (or less), whereas a dictionary attack tries a list of pre-defined words (or a list of commonly used passwords) and their combinations as the password candidates.*
   - *Dictionary attack is relatively faster as there is no need of generating passwords considering all the possibilities of a given character in the password.*

   Steps that can be taken when building passwords to defend from dictionary attacks,

   - *Avoid common words as the passwords.*
   - *Avoid personal information like birthdays, and names, as they can be easily used to launch an attack by making different combinations.*
   - *Avoid using the same password across different services and use unique strong passwords. Use of a password manager can help in this case.*

2. (8 pts) Implement the dictionary attack: `candidate_dict_generator()` Add a screenshot of the modified function.



```python
def candidate_dict_generator(words, suffix):
    counter = 0
    lw = len(words)
    lc = len(suffix)

    # number of candidates = bare words + bare words * suffixes (combinations)
    num_candidates = lw + lw * lc

    # generate candidates
    while True:

        yield get_candidate_dict(counter, words, suffix)

        # stop when all candidates have been generated:
        # counter goes from 0 to num_candidates-1
        if counter >= num_candidates-1:
            return

        counter += 1
```

Figure 2: Screenshot of the modified function

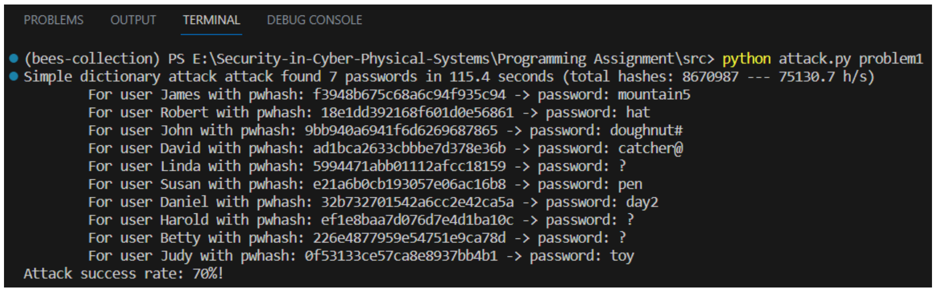
3. (5 pts) What kind of passwords are recovered and for which users compared to Problem 0? Add a screenshot of your terminal output showing the answer. How long would it take to recover the same passwords using the bruteforce attack (Problem 0)? (Justify your answer.)

When comparing with the password recovered from the bruteforce attack, dictionary attack has been able to recover longer passwords for the user, James, John and David.

User	Password
James	mountain5
Robert	hat
John	doughnut#
David	catcher@
Susan	pen
Daniel	day2
Judy	toy

Table 2: Recovered Passwords

- *Time for the attack = 115.4 seconds*
- *Speed of the attack = 75130.7 hashes per second*
- *Attack success rate: 70%*



```
PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

● (bees-collection) PS E:\Security-in-Cyber-Physical-Systems\Programming Assignment\src> python attack.py problem1
● Simple dictionary attack attack found 7 passwords in 115.4 seconds (total hashes: 8670987 --- 75130.7 h/s)
        For user James with pwhash: f3948b675c68a6c94f935c94 -> password: mountain5
        For user Robert with pwhash: 18e1dd392168f601d0e56861 -> password: hat
        For user John with pwhash: 9bb940a6941f6d6269687865 -> password: doughnut#
        For user David with pwhash: ad1bca2633cbbbe7d378e36b -> password: catcher@
        For user Linda with pwhash: 5994471abb01112afcc18159 -> password: ?
        For user Susan with pwhash: e21a6b0cb193057e06ac16b8 -> password: pen
        For user Daniel with pwhash: 32b732701542a6cc2e42ca5a -> password: day2
        For user Harold with pwhash: ef1e8baa7d076d7e4d1ba10c -> password: ?
        For user Betty with pwhash: 226e4877959e54751e9ca78d -> password: ?
        For user Judy with pwhash: 0f53133ce57ca8e8937bb4b1 -> password: toy
  Attack success rate: 70%!
```

Figure 3: Output of the `python attack.py problem1`

The longest recovered password has $l = 9$ characters and they contain special characters from the list (defined in the program) in addition to the alphabet characters, therefore in total we have 26 + 36 possibilities per location in the password.

Total number of possible passwords say P,

$$P = \frac{62^{l+1} - 1}{61} = \frac{62^{9+1} - 1}{61} \approx 1.38 \times 10^{16}$$

Average speed of the attack (this changes runtime-to-runtime and depending on the specifications of the machine used) therefore assume it to be $S = 60 \times 10^3$ hashes per second (an approximate value to the value that observed previously in problem0).

Therefore approximate time in years say Y,

$$Y = \frac{1.38 \times 10^{16}}{60 \times 10^3} \times \frac{1}{365 \times 24 \times 60 \times 60} \approx 7293.25 \; years$$

Brute force attack is not possible on this kind of passwords.

Problem 2: Building a Rainbow table (26 pts)

In this problem, you will perform a Rainbow table attack against the same list of stolen password hashes (`data/simple-dbdump.json`). You will implement (part) of the password lookup rainbow table operation `lookup_rainbow`() in `rainbow.py`. The code to build the rainbow table (`build_rainbow`()) is provided. It will be useful to carefully read the code of `build_rainbow`() and `test_rainbow_attack`() (in `attack.py`).

Use the following to perform the attack:

```
python attack.py problem2
```

1. (2 pts) How do rainbow tables differ from hash-reduce chains? Illustrate with an example.

   In below illustrations, si, hi, pi and ei stands for start, hash, plaintext and end values respectively.

   Hash-reduce chains use the same reduce function (R) throughout the chain. This leads to potential merging of some hash-reduce chains as the reduce functions may not be collision resistant.

   ```
   s1 --H--> h1 --R--> p1 --H--> h2 --R--> e1
   s2 --H--> h3 --R--> p2 --H--> h4 --R--> e2
   s3 --H--> h5 --R--> p3 --H--> h6 --R--> e3
   ```

   Rainbow tables solves this problem by using different reduce functions, k number of different reduce functions, in a chain of length k. In this case a collisions is only posible if chains hit the same value at the same iteration.

   ```
   s1 --H--> h1 --R1--> p4 --H--> h7 --R2--> e4
   s2 --H--> h3 --R1--> p5 --H--> h8 --R2--> e5
   s3 --H--> h5 --R1--> p6 --H--> h9 --R2--> e6
   ```

2. (2 pts) Briefly explain how you determine the length of a chain and the number of chains in a rainbow table.

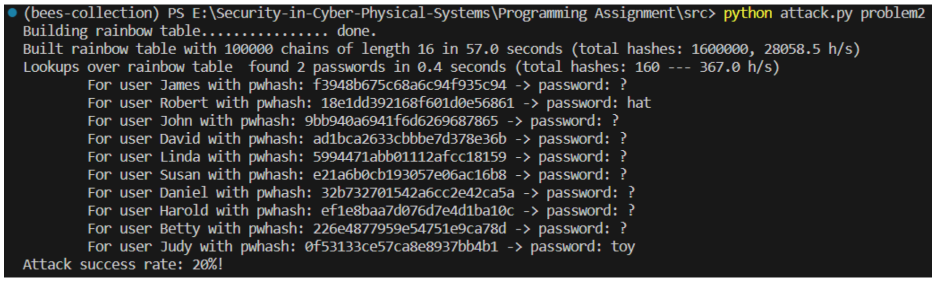
   The length of a chain and the number of chains in a rainbow table represent a trade-off between the time and memory required to launch an attack. The more chains there are, the more disk space is required. The shorter the length of the chain, the faster the attack. Therefore the values must be determined considering the available computing. resources

3. (3 pts) How can you select different reduce functions to be used in different stages of the rainbow table?

   'Different' reduce functions not necessarily mean that they are completely different set of function implementations. Most of the time it is the same reduce function whose output made depend on the iteration (location) of the chain. The iteration is also passed as an argument to the reduce function, which makes sure the same hash does not produce the same reduce value if the iteration are different.

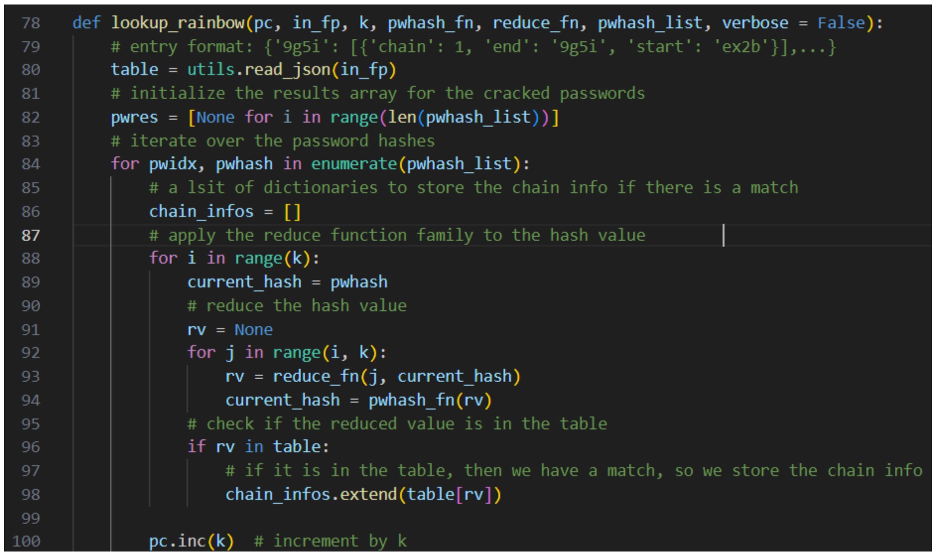
4. (16 pts) Complete the implementation of the rainbow table attack: `lookup_rainbow()`! Test your implementation thoroughly. Your attack should recover the same kind of passwords as for Problem 0. Add screenshots of your terminal output showing the answer and the function implementation.

   - *Time for the attack = 57.0 seconds*
   - *Speed of the attack = 28058.5 h/s for rainbow-table building, 367.0 h/s for the attack*
   - *Attack success rate: 20%*



```
(bees-collection) PS E:\Security-in-Cyber-Physical-Systems\Programming Assignment\src> python attack.py problem2
Building rainbow table................ done.
Built rainbow table with 100000 chains of length 16 in 57.0 seconds (total hashes: 1600000, 28058.5 h/s)
Lookups over rainbow table  found 2 passwords in 0.4 seconds (total hashes: 160 --- 367.0 h/s)
        For user James with pwhash: f3948b675c68a6c94f935c94 -> password: ?
        For user Robert with pwhash: 18e1dd392168f601d0e56861 -> password: hat
        For user John with pwhash: 9bb940a6941f6d6269687865 -> password: ?
        For user David with pwhash: ad1bca2633cbbbe7d378e36b -> password: ?
        For user Linda with pwhash: 5994471abb01112afcc18159 -> password: ?
        For user Susan with pwhash: e21a6b0cb193057e06ac16b8 -> password: ?
        For user Daniel with pwhash: 32b732701542a6cc2e42ca5a -> password: ?
        For user Harold with pwhash: ef1e8baa7d076d7e4d1ba10c -> password: ?
        For user Betty with pwhash: 226e4877959e54751e9ca78d -> password: ?
        For user Judy with pwhash: 0f53133ce57ca8e8937bb4b1 -> password: toy
Attack success rate: 20%!
```

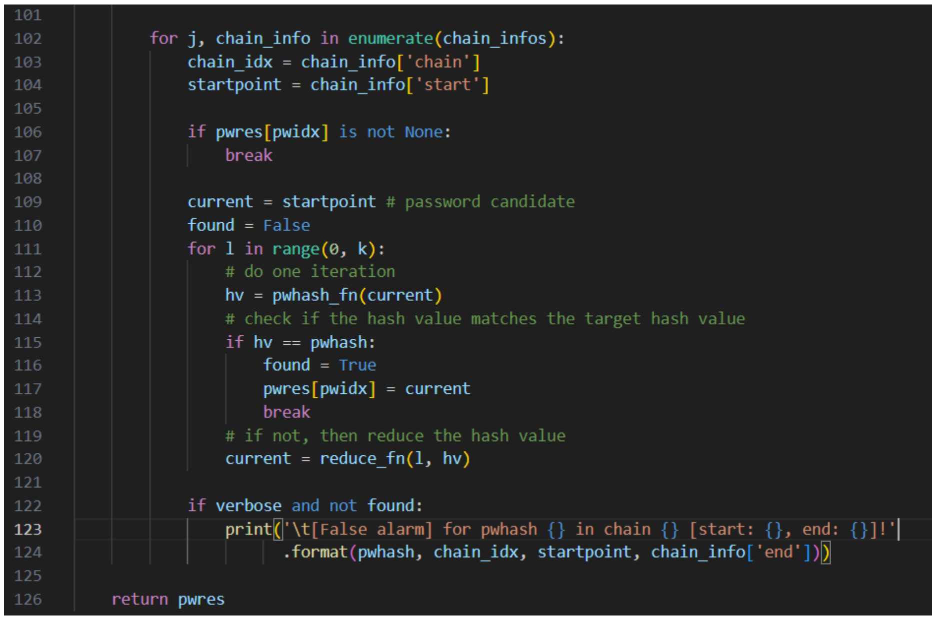
Figure 4: Output of the `python attack.py problem2`



```python
def lookup_rainbow(pc, in_fp, k, pwhash_fn, reduce_fn, pwhash_list, verbose = False):
    # entry format: {'9g5i': [{'chain': 1, 'end': '9g5i', 'start': 'ex2b'}],...}
    table = utils.read_json(in_fp)
    # initialize the results array for the cracked passwords
    pwres = [None for i in range(len(pwhash_list))]
    # iterate over the password hashes
    for pwidx, pwhash in enumerate(pwhash_list):
        # a lsit of dictionaries to store the chain info if there is a match
        chain_infos = []
        # apply the reduce function family to the hash value
        for i in range(k):
            current_hash = pwhash
            # reduce the hash value
            rv = None
            for j in range(i, k):
                rv = reduce_fn(j, current_hash)
                current_hash = pwhash_fn(rv)
            # check if the reduced value is in the table
            if rv in table:
                # if it is in the table, then we have a match, so we store the chain info
                chain_infos.extend(table[rv])

        pc.inc(k)  # increment by k
```

Figure 5: Implementation of the part 1 of `lookup_rainbow()` function



```python

        for j, chain_info in enumerate(chain_infos):
            chain_idx = chain_info['chain']
            startpoint = chain_info['start']

            if pwres[pwidx] is not None:
                break

            current = startpoint # password candidate
            found = False
            for l in range(0, k):
                # do one iteration
                hv = pwhash_fn(current)
                # check if the hash value matches the target hash value
                if hv == pwhash:
                    found = True
                    pwres[pwidx] = current
                    break
                # if not, then reduce the hash value
                current = reduce_fn(l, hv)

            if verbose and not found:
                print('\t[False alarm] for pwhash {} in chain {} [start: {}, end: {}]!'
                      .format(pwhash, chain_idx, startpoint, chain_info['end']))

    return pwres
```

Figure 6: Implementation of the part 2 of `lookup_rainbow()` function

5. (3 pts) What is a false alarm? Explain why it occurs and what we can do to minimize false alarms in rainbow tables.

    False alarms are instances,
    1. We get an end point that is available in the rainbow table through reduce-hash procedure for the hash of a given password,
    2. However when we reconstruct the chain using the starting point of the corresponding end point, we do not hit the required hash of the given password.

    The reason for this, is reduce functions are not collision resistant and different hashes can reduce to the same value.

Problem 3: Understanding Rainbow tables (32 pts)

In this problem, you will run the Rainbow table attack on a larger database of stolen password hashes (`data/complex-dbdump.json`). Unlike for previous problems, in this step all passwords consist of 4 alphanumeric characters — lowercase only.

Use the following command to run the attack:

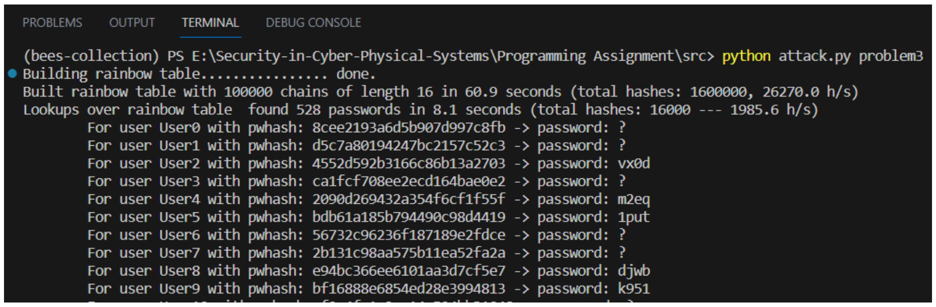
```
python attack.py problem3
```

Let n denote the total number of possible passwords, m denote the number of chains in the rainbow table, and k denote the length of each chain.

1. (3 pts) What is the success rate of the attack (percentage of passwords recovered)? Why is it not 100%?

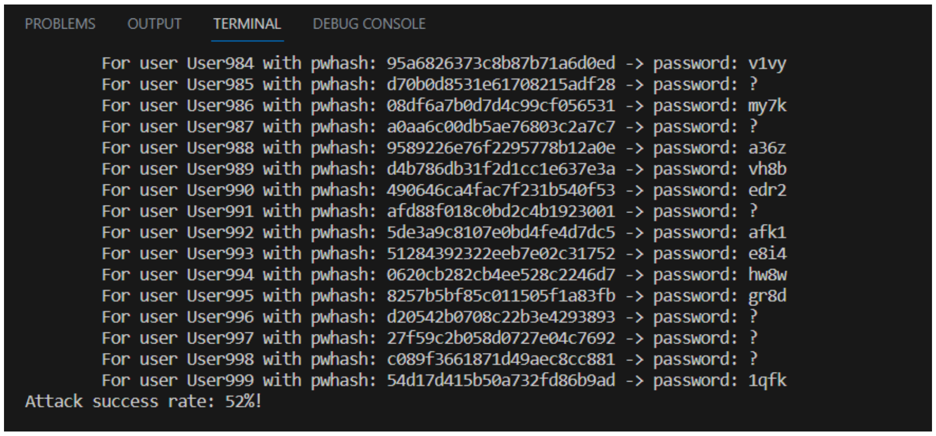
   Success rate = 52%

   success of a rainbow table depends on several factors such as the size of the rainbow table built. In addition to that chances are there that some passwords have been defensed against rainbow-table attack using a salt value.



```
(bees-collection) PS E:\Security-in-Cyber-Physical-Systems\Programming Assignment\src> python attack.py problem3
Building rainbow table................ done.
Built rainbow table with 100000 chains of length 16 in 60.9 seconds (total hashes: 1600000, 26270.0 h/s)
Lookups over rainbow table  found 528 passwords in 8.1 seconds (total hashes: 16000 --- 1985.6 h/s)
        For user User0 with pwhash: 8cee2193a6d5b907d997c8fb -> password: ?
        For user User1 with pwhash: d5c7a80194247bc2157c52c3 -> password: ?
        For user User2 with pwhash: 4552d592b3166c86b13a2703 -> password: vx0d
        For user User3 with pwhash: ca1fcf708ee2ecd164bae0e2 -> password: ?
        For user User4 with pwhash: 2090d269432a354f6cf1f55f -> password: m2eq
        For user User5 with pwhash: bdb61a185b794490c98d4419 -> password: 1put
        For user User6 with pwhash: 56732c96236f187189e2fdce -> password: ?
        For user User7 with pwhash: 2b131c98aa575b11ea52fa2a -> password: ?
        For user User8 with pwhash: e94bc366ee6101aa3d7cf5e7 -> password: djwb
        For user User9 with pwhash: bf16888e6854ed28e3994813 -> password: k951
```

Figure 7: Launching the attack using the command `python attack.py problem2`



```
        For user User984 with pwhash: 95a6826373c8b87b71a6d0ed -> password: v1vy
        For user User985 with pwhash: d70b0d8531e61708215adf28 -> password: ?
        For user User986 with pwhash: 08df6a7b0d7d4c99cf056531 -> password: my7k
        For user User987 with pwhash: a0aa6c00db5ae76803c2a7c7 -> password: ?
        For user User988 with pwhash: 9589226e76f2295778b12a0e -> password: a36z
        For user User989 with pwhash: d4b786db31f2d1cc1e637e3a -> password: vh8b
        For user User990 with pwhash: 490646ca4fac7f231b540f53 -> password: edr2
        For user User991 with pwhash: afd88f018c0bd2c4b1923001 -> password: ?
        For user User992 with pwhash: 5de3a9c8107e0bd4fe4d7dc5 -> password: afk1
        For user User993 with pwhash: 51284392322eeb7e02c31752 -> password: e8i4
        For user User994 with pwhash: 0620cb282cb4ee528c2246d7 -> password: hw8w
        For user User995 with pwhash: 8257b5bf85c011505f1a83fb -> password: gr8d
        For user User996 with pwhash: d20542b0708c22b3e4293893 -> password: ?
        For user User997 with pwhash: 27f59c2b058d0727e04c7692 -> password: ?
        For user User998 with pwhash: c089f3661871d49aec8cc881 -> password: ?
        For user User999 with pwhash: 54d17d415b50a732fd86b9ad -> password: 1qfk
Attack success rate: 52%!
```

Figure 8: Output of the `python attack.py problem2`

2. (4 pts) [Experimental] Keeping $m \cdot k$ constant, run the attack with m = 25000, 50000, 100000, 200000, 400000 and $k = 64, 32, 16, 8, 4$. (You can modify m and k in the `main()` function of `attack.py`.)

   Record the success rate of the attack (p) and write it in the table below.

m	*25000*	*50000*	*100000*	*200000*	*400000*
k	*64*	*32*	*16*	*8*	*4*
p	*54%*	*53%*	*52%*	*56%*	*55%*

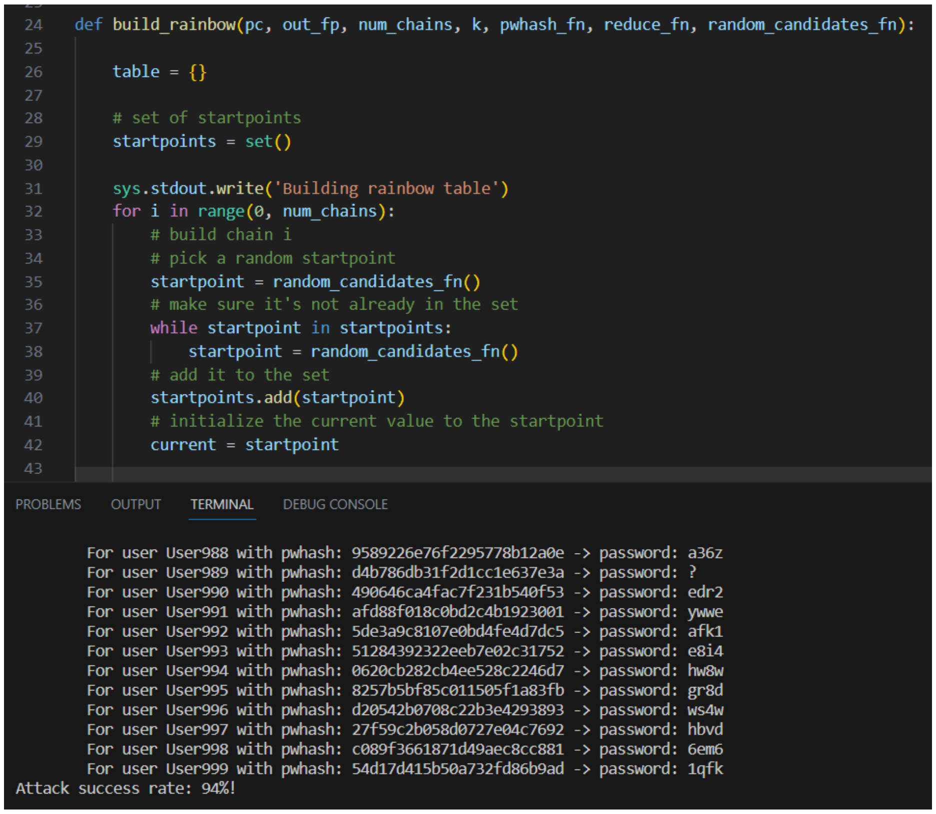
3. (6 pts) Consider the case $k = 1$. Observe experimentally that the success rate of the attack is lower than $\frac{m \cdot k}{n}$. Why is that? To fix it, change the implementation of `build_rainbow()` to ensure that each chain gets a unique startpoint. (Be careful to ensure that `build_rainbow()` always terminates no matter the values of n, m, and k.)

    Success rate for k = 1, is 60%

    Note that, unlike for previous problems, in this step all passwords consist of 4 alphanumeric characters — lowercase only, therefore total number of passwords $n = 36^4$.

    Expected success rate $= \frac{m \cdot k}{n} = \frac{1600000 \cdot 1}{36^4} = 95.26\%$

    *A possible reason for this experimental success rate to be lower than the expected success rate is that, **start values of chains (password candidates) are randomly generated. Therefore, all possible passwords are not guaranteed to be captured during the table creation**. That means there can be chins with the same starting value.*



```
def build_rainbow(pc, out_fp, num_chains, k, pwhash_fn, reduce_fn, random_candidates_fn):

    table = {}

    # set of startpoints
    startpoints = set()

    sys.stdout.write('Building rainbow table')
    for i in range(0, num_chains):
        # build chain i
        # pick a random startpoint
        startpoint = random_candidates_fn()
        # make sure it's not already in the set
        while startpoint in startpoints:
            startpoint = random_candidates_fn()
        # add it to the set
        startpoints.add(startpoint)
        # initialize the current value to the startpoint
        current = startpoint

```

```
        For user User988 with pwhash: 9589226e76f2295778b12a0e -> password: a36z
        For user User989 with pwhash: d4b786db31f2d1cc1e637e3a -> password: ?
        For user User990 with pwhash: 490646ca4fac7f231b540f53 -> password: edr2
        For user User991 with pwhash: afd88f018c0bd2c4b1923001 -> password: ywwe
        For user User992 with pwhash: 5de3a9c8107e0bd4fe4d7dc5 -> password: afk1
        For user User993 with pwhash: 51284392322eeb7e02c31752 -> password: e8i4
        For user User994 with pwhash: 0620cb282cb4ee528c2246d7 -> password: hw8w
        For user User995 with pwhash: 8257b5bf85c011505f1a83fb -> password: gr8d
        For user User996 with pwhash: d20542b0708c22b3e4293893 -> password: ws4w
        For user User997 with pwhash: 27f59c2b058d0727e04c7692 -> password: hbvd
        For user User998 with pwhash: c089f3661871d49aec8cc881 -> password: 6em6
        For user User999 with pwhash: 54d17d415b50a732fd86b9ad -> password: 1qfk
Attack success rate: 94%!
```

Figure 9: Modifications to make sure there are no duplicate start points

As illustrated in the Figure 9, once we avoid the duplicate start points the success rate is 94%.

4. (4 pts) [Theory] Explain the difference between hash and reduce functions and give 3 examples of possible reduce functions.

   A hash function is a mathematical function which converts a given input message to a fixed-length string of bytes. The resultant representation is unique for a given input and small change in the input can cause a drastic change in the output (avalanche effect). In addition a hash functions are, 1. pre-image resistance, 2. second pre-image resistance and collision resistance.

   In contrast, reduce function is not an actual reverse of a hash value. It takes in a hash value and maps it to a value in the space of possible passwords through some mechanism. This mechanism yields a unique value for a given has. However, they are not collision resistant, meaning two different hash values can map to the same value. Possible examples of reduce functions are given below.

   - ***Truncation****: truncates the hash value to a fixed length and maps it to a value in the password space.*
   - ***Arithmetic operations****: performs arithmetic operations like modulo, integer division on the hash value or variant of it to map it to a value in the password space.*
   - ***Bit extraction****: extracts certain bits from the hash value and uses them as an index into the password space.*

5. [Theory] Assume this password-cracking algorithm is implemented on a memory-limited system. Assume that a chain can be stored using 8 bytes (4 bytes for the startpoint, 4 bytes for the endpoint). Suppose we set m and k such that $n = m \cdot k$. Consider the system can compute 1000 hashes per second and 1000 reduction functions per second. (Assume time taken for other instructions is negligible when compared with hashing and reduction).

   (a) (2 pts) What is the amount of memory required to store the rainbow table?

   Memory required = Memory per chain $\times$ Number of chains = 8 Bytes $\times$ m = $8m$ Bytes

   (b) (3 pts) What is the expected computational complexity of a lookup?

   Please note that, when calculating the time complexity of the lookup for a single password, the actual implementation of the function `lookup_rainbow()` *is used.*

   It consists of two main phases,

   i. ***Finding the chains which can potentially include the password.***

   This part of the function is shown in the Figure 5. It consists of one nested for loop which computes hashing + reducing, $\frac{k}{2} \times (k+1)$ times. Finding a chain in the dictionary object has constant time complexity and therefore neglected.

   ii. ***Rebuilding the chains to find the password.***

   This part of the function is shown in the Figure 6. It rebuilds the chain for each potential chain match and check whether the password is in a given chain. At most we can have m potential chains (total chains) and for each chain we have k number of hashing + reducing. Therefore this part of the function computes hashing + reducing, $m \times k$ times.

Therefore the total time complexity of the entire lookup for a single password is,

$$O\left(\frac{k}{2} \times (k+1)\right) + O\left(m \times k\right) = O\left(k^2\right) + O\left(m.k\right)$$

Since we keep the $n = m.k$ constant, The actual time complexity of the lookup is $O(k^2)$.

(c) (4 pts) Determine the optimum value for k such that it uses the lowest memory space to store the rainbow table.

Memory required in Bytes $= 8m$
Replacing m using the equation $n = m \cdot k \rightarrow m = \frac{n}{k}$

Memory $= 8 \times \frac{n}{k}$

As it can be seen to minimize the memory usage, we need to increase the chain length. Therefore we need to select the largest possible k, which respects all the time complexity constraints as well. Therefore, k value becomes a trade-off, which must be calculated experimentally.

6. (6 pts) [Experimental] Keeping $m \cdot k$ constant, run the attack for $m = 25000$, 50000, 100000, 200000, 400000 and $k = 64, 32, 16, 8, 4$. Plot the time for a lookup in each case. How does this compare to your answer to Problem 3, part 5(b)?

m	*25000*	*50000*	*100000*	*200000*	*400000*
k	*64*	*32*	*16*	*8*	*4*
p	*54%*	*53%*	*53%*	*57%*	*58%*
t/lookup	*0.11972*	*0.03027*	*0.00837*	*0.00207*	*0.00059*

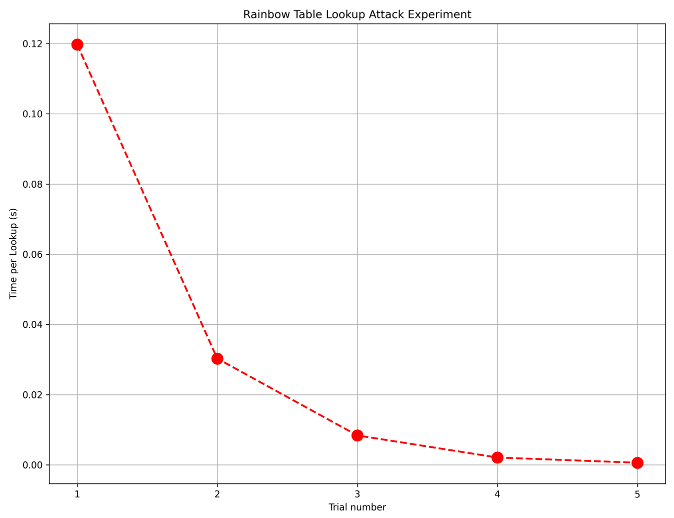


Figure 10: Plot of the time for a lookup in each case

When this plot is compared with the answer to Problem 3, part 5(b) the expected time complexity $O(k^2)$ agrees with the experimental results. That is when the k is decreased, the time for a lookup has reduced exponentially.

Problem 4: Bob's custom password hash (14 pts + [bonus] 10 pts)

Your classmate Bob is in charge of the website of E-Club, ENTC. To implement authentication on the website, Bob (who has not taken EN4720) has decided to roll his own crypto. He has implemented a custom password hash function. Given a password p of even length, p is split into two parts of equal lengths p_1, p_2. The password hash is then computed as:

$$H_l(p_1)||H_l(p_2)||H_l^i(s||c),$$

where s is a random salt, c is a constant string, i and l are positive integers. Here: $H_l(x)$ denotes the first l bytes of the hash of x. This password hash is implemented as `bobs_custom_pw_hash()` in `crypto.py`.

For this problem, you will implement your own attack against Bob's custom password hash scheme. For this you will use `data/bobX-dbdump.json` as database of password hashes.

Use the following command to run your attack:

```
python attack.py problem4
```

1. (2 pts) Is it a good idea to design your own crypto like Bob did in this case? Explain why or why not?

   It is not advised to design your own crypto, especially if you are not an expert in the field of cryptography.

   According to the Schneier's law "Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break."

   Therefore the best option is to go for a publicly available well-established crypto which has undergone extensive scrutiny.

2. (3 pts) Give your comments on the salting mechanism used by Bob in his custom crypto. Does his method serve the purpose of salting? Explain your answer.

   *To serve the purpose of making it harder to crack the password, **the salt must be concatenated with the password before hashing it.***

   *However, what Bob has done is meaningless. Because, **the two parts of the password and salt are hashed separately and then concatenated.** Still the real password is not contaminated by the salt.*

   Password can still be recovered by attacking only the $H_l(p_1)||H_l(p_2)$ part of the hash without considering the hash part coming from the salt.

3. (3 pts) Bob's friend Alice says the splitting of the password into 2 equal parts and hashing them separately will make the life of an attacker difficult to launch a bruteforce attack. Do you agree with this statement? Give reasons.

   Disagree with the statement.

   Splitting the password in to 2 equal parts, will not make the life of an attacker difficult to launch a bruteforce attack. Because now the length of the password has become half and possible size of the password space has reduced drastically.

   As an example let's say password is 8 characters and lowercase alphanumeric.

   - *If the password is not split, total possible passwords* $= 36^8$
   - *however, after splitting total possible passwords* $= 36^4$*, and each part can be brute forced separately. This would be much faster than brute-forcing the entire password.*

4. (3 pts) Is it a good idea to use the first l bytes of the hashes as in Bob's crypto? Explain your answer giving reasons.

   It is a bad idea.

   One of the main properties of a hash value is its collision resistance, which makes it very difficult for an attacker to find two input values with the same hash. For this property to hold it hash should be of full length.

   However, considering the first l bytes will reduce number of possible hashes, as now we do not need to consider the full hash length. This makes it easier for an attacker to find a collision.

5. (3 pts) Suggest a suitable method to make this custom password hash scheme difficult to crack by the attackers.

   The fundamental weaknesses of Bob's password scheme are follows.

   - *Salt is not actually concatenated (applied) to the passwords; it is separate.*
   - *Password has been split into two parts making it easy to brute force them separately.*
   - *Only the first l bytes of the hashes have been considered to build the final password hash.*

   Hashing it again will not address this fundamental weaknesses. Therefore, in order to make the scheme stronger, we have to directly address the mentioned weaknesses by doing followings.

   - *Make it a requirement to have a minimum length (at least 8) to the bare password and add salt directly to the password parts to make the brute-forcing difficult. Eg. $p1 + s \geq 8$ & $p2 + s \geq 8$ will be a good choice.*
   - *Increase the number of bytes (l) that we consider when building the password hash. This will make it more collision resistant.*

6. ([bonus] 10 pts) Implement the best attack you can think of. Place your code in the provided placeholders in `bobs_custom_pwhash_attack()` (`attack.py`). You will be evaluated based on the performance of your attack on passwords of varying length. You can change the size of the targets passwords by changing the value of `pw_length` in the `main()` function of `attack.py`.

   Hint: the fastest attack is neither the naive bruteforce attack nor the one based on rainbow tables.

   Explain how your attack works. How fast is it on passwords of length 4, 6, and 8?

   Not implemented.