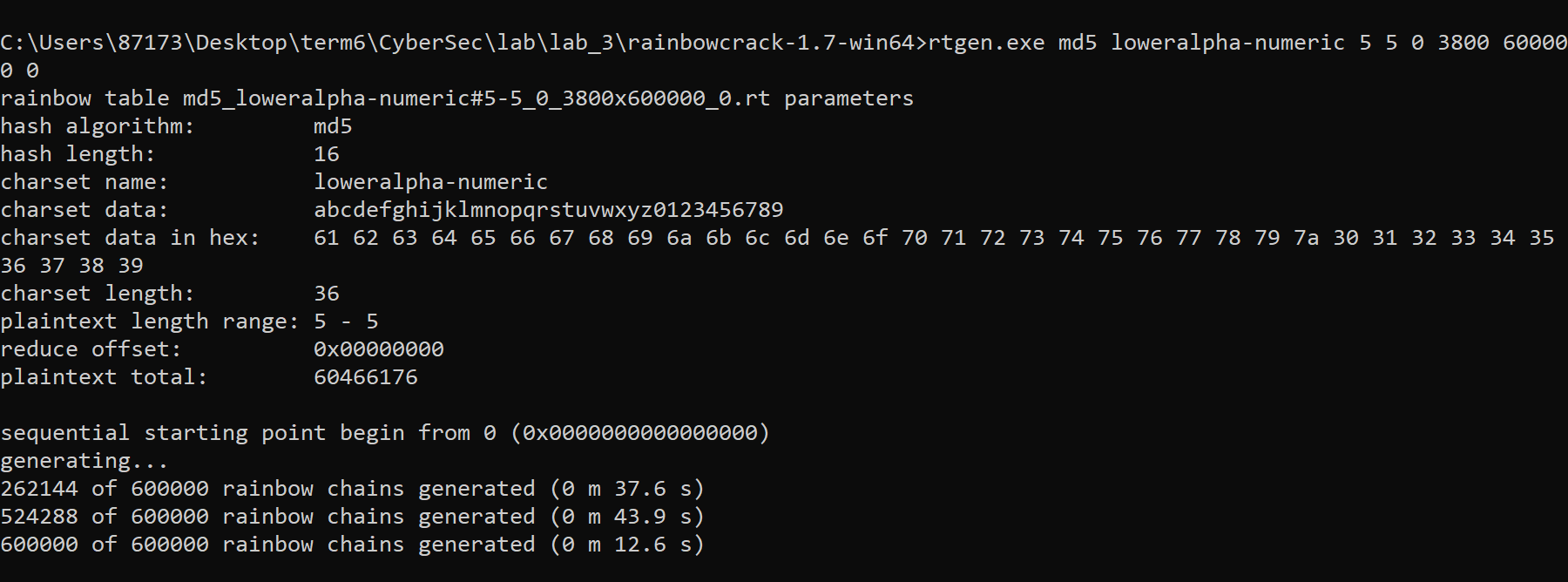
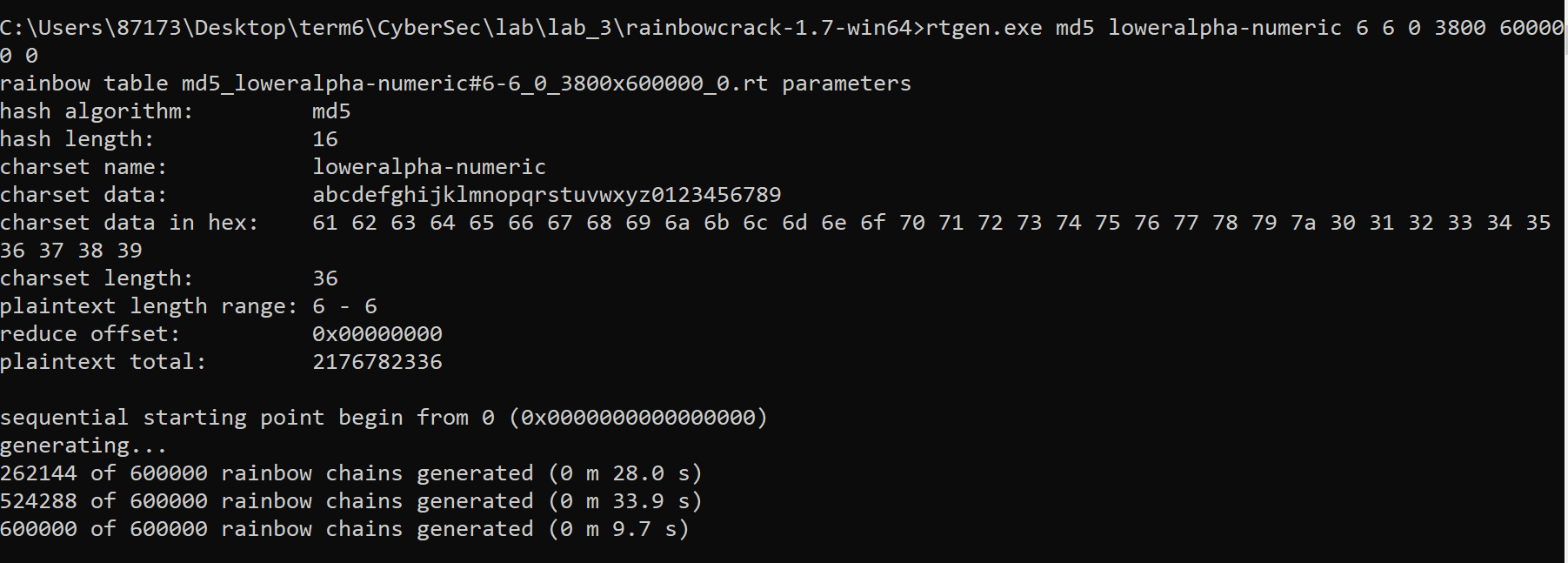
**5. salting**

**- Compare the timing of the new table generation vs the previous values**

The time of generating the new table is shorter than the one of the previous table, as shown below:

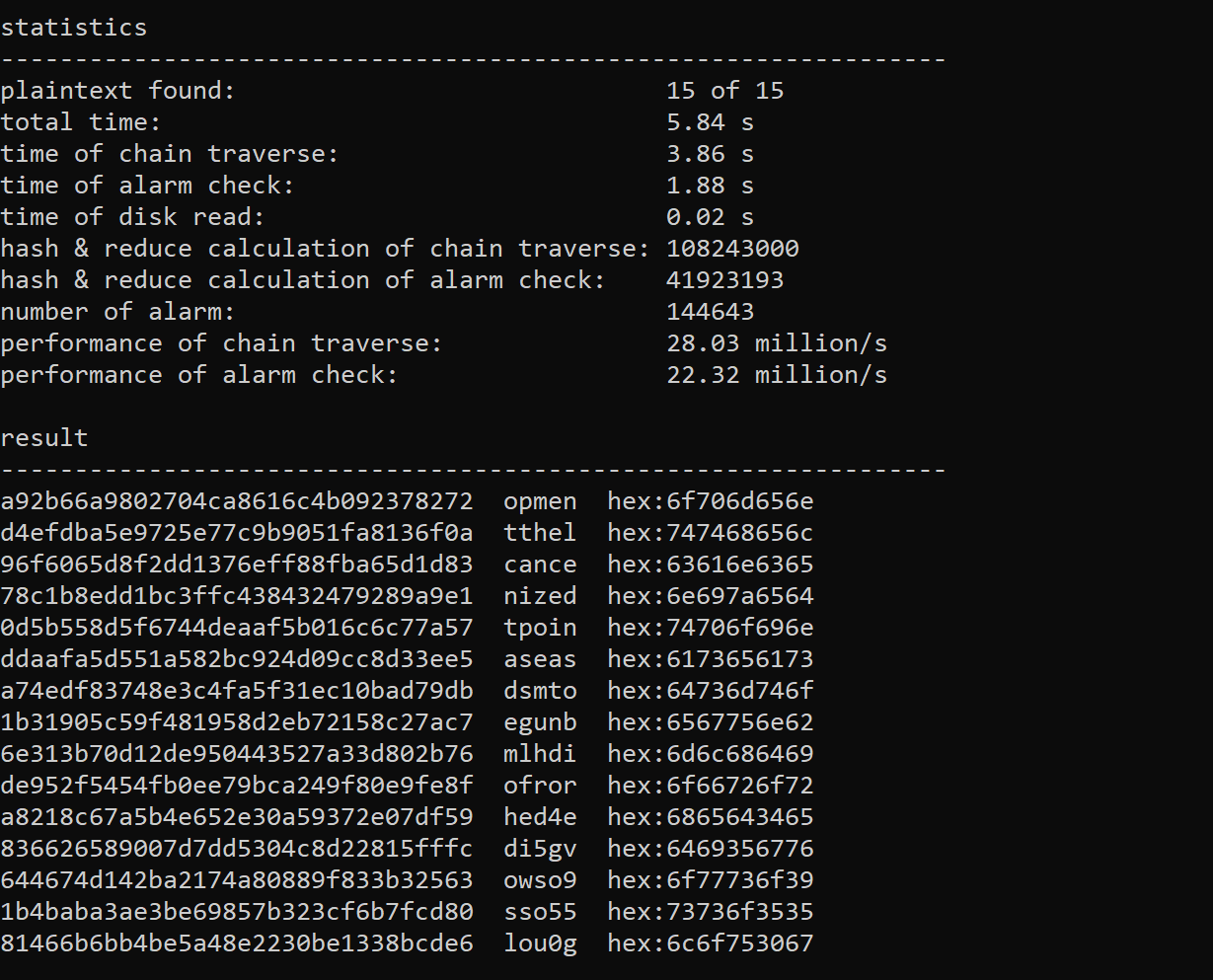
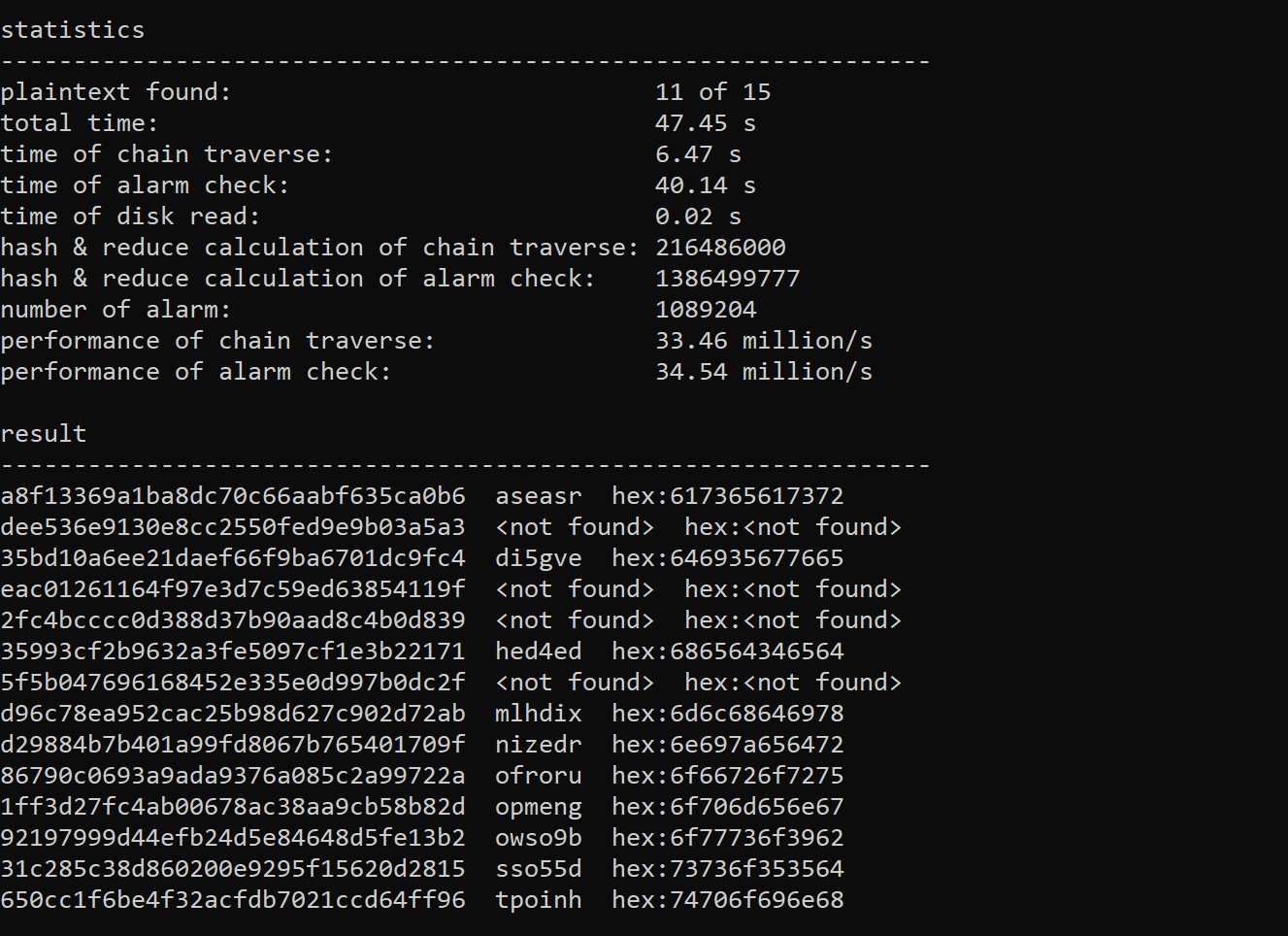


(time for generating the previous table)



(time for generating the new table)

**- Compare the timing of the new table lookup vs the previous values**

The time of cracking the salted hashes in the new table (47.45s) is longer than the one of cracking unsalted hashed in the previous table (5.84s). As shown below:  

(time for looking up unsalted hashes in previous table) (time for looking up salted hashes in new table)

**- Explaining the differences between salted and non-salted rcrack strategies and comparing the times**

The time of cracking the salted hashes in the new table (47.45s) is longer than the one of cracking unsalted hashed in the previous table (5.84s).

Firstly, as we know, rainbow table is a compromise between pre-computation and low memory usage, thus it is kind of a list of pre-computed hashes of commonly-used password, which means to save the memory space, it will not cover the hashes of all complex passwords.

Now considering the non-salted plaintext, there is a high chance that its hash value is stored in the rainbow table. Therefore, it will take a short time to recover the plaintext.

However, if the plaintext is added with a random character, which makes it becomes less commonly-used, then it is probably not stored in the rainbow table (since rainbow table only contains the commonly-used ones). Therefore, a longer time will be required for looking up the hash values in the rainbow table and for some rare ones ( which means the salt is random enough), it may even not be found in the rainbow table.

**6. Hash breaking competition**

Firstly, I write a python script to crack the hashes in using brute force way.

The following is the explanation of my logic.

1. **Define possible characters**

Firstly, I defined all the possible characters that will appear in the plaintext, which are all ascii characters (printable characters).

1. **Calculate permutations**

Once we know the characters that will appear in the plaintext, we can list the permutation of these characters by using the itertools package.

However, in this case, we have no information about the length of the plaintext. So, I do the permutation with parameter repeat=1, 2, …, n where n is the total number of printable characters.

For example, if the possible characters is ‘abc’:

With repeat=1, permutation\_list = ['a', 'b', 'c']

With repeat=2, permutation\_list = ['aa', 'ab', 'ac', 'ba', 'bb', 'bc', 'ca', 'cb', 'cc']

….

By doing this, we can get all the permutations with these printable characters.

In my code, I put it in the following way to calculate the permutations.

possible\_permutation = [''.join(x) for i in range(0, 5) for x in product(char\_in\_password, repeat=i)]

However, there will be memory error if I set upper bound of the repeat value to be larger than 5. (Ideally it should be able to calculate all the possible permutations if there is no memory error.)

1. **Hash the permutations one by one and check if the hashed value is in hashes.txt**

Once I get the list of permutations, I will hash each of permutation in the list and compared the hashed values with the ones in hashes.txt. If the hashed value is in hashes.txt, the hashed value and that permutation will be appended into the csv file.

Due to the memory error, I am only able to crack the hashes which the plaintexts only contain 5 or less character. The rest ones I use the online hash cracker to get the plaintexts.