CPSC 42800 Project 6: Multiprocessing Rainbow Table

Table of Contents

1. Overview

For this project you will write multi-processing Python code to generate a *rainbow table*, and use your table to attempt to crack passwords in a simulated password database.

In real-world digital forensic investigations, it may be necessary to obtain access to a password-protected account or device. Completing this project will give you just that type of experience.

This project is all about performance. An important component of the project is measuring and reporting the speedups (if any) that you obtain through the use of multiprocessing.

2. Detailed Requirements

The project consists of 3 major components, which need to be completed in order.

2.1. Getting the book code to run with SHA-256 (40 points)

Your first job is to turn the code printed on pages 280-282 and 283-286 of *Python Forensics* into **two** working Python 3 programs. The first is a single-core version, and the second a multi-core version, of a password hash (rainbow table) generator. Name the first file `onecore.py` and the second `fourcore.py`.

There is one change in functionality you are required make to the book programs: They must use the **SHA-256** hash function instead of MD5. The MD5 hash function is obsolete for security purposes. This should be a straightforward change in the code, since both hash functions are offered by the Python *hashlib* library.

Each program will perform the same operations, which is to

- Generate a set of passwords of 2, 3, 4, and 5, characters;
- Create SHA256 hashes of all the passwords;
- Store the hashes in files;
- Run test code that loads the output tables into a dictionary and performs a lookup on a sample hash value;
- Compute the elapsed time for these operations.

There may be bugs or incompatibilities in the book's code; it is your responsibility to track these down and fix them. This will require you to carefully **read and understand the code** to determine its intended operation. Specifically, you will have to deal with Python 3 string encoding issues similar to what we have seen in class.

You must *test your program sufficiently* to ensure that it works with all options and is as bug-free as possible. This includes inspecting the output files and log files.

Hint: In changing the hash function used from MD5 to SHA-256, you will need to update the sample hash used in the test code to avoid an error. The SHA256 hash of the sample password “#I#$$” is:

```
1dbdfd6de15b28f247ec7e1ec571b9f49098b82a6be400baa0fe0e44aedc4e1c
```

2.2. Code Improvements (40 points)

You are to design and implement the following extensions to the code given in the book. Both versions of the program should have the extensions.

Two-process version (10 points). Create a third version of the multi-core program that uses only *two* processes. Besides the obvious change, this program will also need to alter how it splits up the space of all possible passwords among two processes. You should strive to do this so that the workload is evenly balanced between the two processes. Name this program file `twocore.py`.

Cracking a given password file (30 points). Create a separate script file, `crackfile.py`, for attempting to crack a password database file using the output table files generated by the above programs.

This script should take one command-line argument that is the file name of a plaintext password database file. The format of this file is that each line contains a username, followed by a colon, followed by a hexadecimal SHA256 password hash. Here is an example:

```
bsmith:c9efc14482fc976da42c8f2af73551c31af35bcfac5267fbdca395b79a05d0fc
```

The file can have any number of lines of this form.

The script should load a rainbow table file from disk (the filename of the rainbow table can be hard-coded) and create a dictionary from it, the same way the test code in the book programs does. Then, it should read the specified password database file from disk and attempt to find a password that matches each hash value contained in the file. The program should print out the (username, password) pair for any successfully cracked passwords. You may assume the salt is the same as the hard-coded value in the program.

This program does not need to be parallelized. **The cracking program should not recompute any hashes!** The point of a rainbow table is to precompute all hashes *once*, then use it as a database to crack quickly in the future. Also, this program should use dictionary lookup to find the matching hash (if any) quickly, rather than linearly searching through all the hashes. Your finished `crackfile.py` should finish searching for all five hashes within a few seconds.

You are provided with a sample password database, “shadow.txt”, to try to crack. Should the generated rainbow tables not suffice to crack all the passwords in the file, you are welcome to try extending the set of passwords hashed to crack more passwords. :)

2.3. Performance Investigation and cracking results (20 points)

You should compare the performance of the single-core, two-core, and four-core versions of the table generator program. On a single machine, for each of the three programs, run five trials and record the times printed out (a spreadsheet would be good for this). Then take the average time for each of the three programs. The report you turn in should contain both the 15 individual times and the 3 averages. Also, provide a record of the the three averages converted to units of **passwords per second**, to give a better understanding of the results.

In your report, record the processor, amount of RAM, OS, and disk type used by your test machine. Write a paragraph with your hypotheses on what made the performance numbers be what they were on the hardware and OS you used.

To make the quality of your measurements as high as possible, you should attempt to run your tests in a “clean” environment, with as few additional programs running on the machine as possible, and giving the machine sufficient time to “settle” after booting or waking up before running your tests.

Finally, your report should also contain the results of attempting to crack passwords in the file (you don't need to time that part.)

3. What to Submit

You are to submit on Blackboard a single zipped archive, containing:

- The python source code for the programs (four in all). DO NOT include any bytecode (.pyc) files.
- Your report file with the results of your timing experiments and password file cracking attempt.

4. Concluding remarks

1. Start early.
2. If your program isn't working, *make it tell you why*. Programming is 20% typing and 80% detective work. As an analogy to criminal investigation, imagine that your program is an uncooperative witness and your job is to extract the evidence you need from them.
3. If the above fails, ask me for help. As always, I am available to help with debugging your code.
4. Have fun!

Created: 2021-12-01 Wed 08:31
Validate