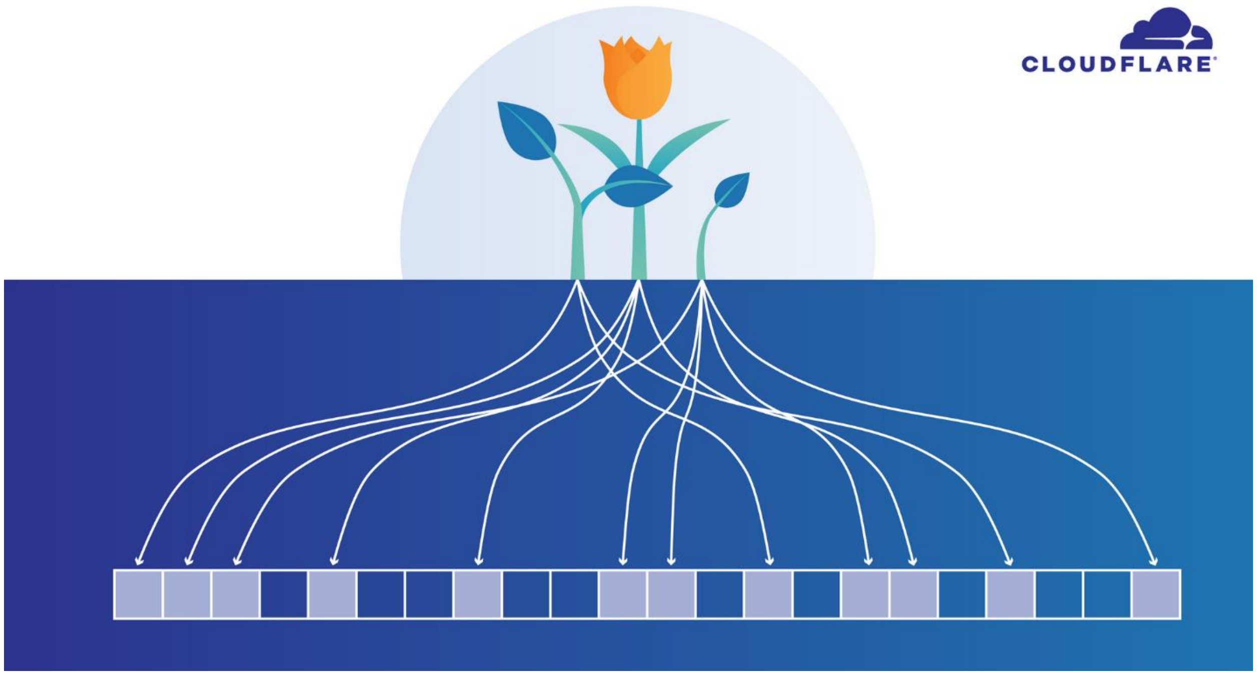
Double Bloom Filter & Password Strength

MMN13 - Programming assignment



Background

Bloom filter

A Bloom filter is a set-like data structure that is highly efficient in its use of space. It only supports two operations: insertion and membership querying. Unlike a normal set data structure, a Bloom filter can give incorrect answers. If we query it to see whether an element that we have inserted is present, it will answer affirmatively. If we query for an element that we have *not* inserted, it *might* incorrectly claim that the element is present.

For many applications, a low rate of false positives is tolerable. For instance, the job of a network traffic shaper is to throttle bulk transfers (e.g. BitTorrent) so that interactive sessions (such as **ssh** sessions or games) see good response times. A traffic shaper might use a Bloom filter to determine whether a packet belonging to a particular session is bulk or interactive. If it misidentifies one in ten thousand bulk packets as interactive and fails to throttle it, nobody will notice.

The attraction of a Bloom filter is its space efficiency. If we want to build a spell checker, and have a dictionary of half a million words, a set data structure might consume 20 megabytes of space. A Bloom filter, in contrast, would consume about half a megabyte, at the cost of missing perhaps 1% of misspelled words.

Behind the scenes, a Bloom filter is remarkably simple. It consists of a bit array and a handful of hash functions. We'll use *k* for the number of hash functions. If we want to insert a value into the Bloom filter, we compute *k* hashes of the value, and turn on those bits in the bit array. If we want to see whether a value is present, we compute *k* hashes, and check all of those bits in the array to see if they are turned on.

The Bloom Filter's algorithm work flow is defined by turning on the bits when implementing the hash function according to the result.

Since the number of bits presented in the Bloom Filter is finite, at some point there will be parallels between the significant bits that are turned on as a result of valid inputs that are different from the values in the dictionary.

This problem is called the FALSE POSITIVE.

If we were to define the entries in the dictionary as M, the number of elements (bits) as N and the position in the array as K, we can define the probability of getting a FP:

$$\sum_t \Pr(q=t)(1-t)^k \approx (1-E[q])^k = \left(1-\left[1-\frac{1}{m}\right]^{kn}\right)^k \approx \left(1-e^{-kn/m}\right)^k$$

In order to solve this problem, we suggest taking two sets of data, applying a bloom filter to each one and then combining them into one filter in order to find out and prevent multiplications between both sets.

The solution, though costs twice as much for the dictionaries to be implemented, however it doubles the possible password pool size and reduces by half the probability of receiving a FP.

Password strength

Password strength is a measure of the effectiveness of a password against guessing or brute-force attacks. In its usual form, it estimates how many trials an attacker who does not have direct access to the password would need, on average, to guess it correctly. The strength of a password is a function of length, complexity, and unpredictability.

Using strong passwords lowers overall risk of a security breach, but strong passwords do not replace the need for other effective security controls. The effectiveness of a password of a given strength is strongly determined by the design and implementation of the factors (knowledge, ownership, inherence). The first factor is the main focus in this article.

The rate at which an attacker can submit guessed passwords to the system is a key factor in determining system security. Some systems impose a time-out of several seconds after a small number (e.g. three) of failed password entry attempts. In the absence of other vulnerabilities, such systems can be effectively secured with relatively simple passwords. However, the system must store information about the user passwords in some form and if that information is stolen, say by breaching system security, the user passwords can be at risk.

In this work we compute a "score" for a given password using the following rules:

Top Requirement (will also be used in the calculation of the score, see below table for details):

- Password Length >= 8
- Contains Uppercase Letters (A-Z)
- Contains Lowercase Letters (a-z)

- Contains Digits (0-9)
- Contains Symbols (asci is not of a letter and not of a number)

Penalties (that will cause reductions in score):

Condition	Action
IF Password is **all letters**	**Reduce** Password length
IF Password **is all digits**	**Reduce** Password length
IF Password has repeated characters	**Reduce** (Number of repeated characters * (Number of repeated characters -1))
IF Password has consecutive uppercase letters	**Reduce** (Number of consecutive uppercase characters * 2)
IF Password has consecutive lowercase letters	**Reduce** (Number of consecutive lowercase characters * 2)
IF Password has consecutive digits	**Reduce** (Number of consecutive digits * 2)
IF Password has sequential letters	**Reduce** (Number of sequential letters * 3) E.g.: ABCD or DCBA.
IF Password has sequential digits	**Reduce** (Number of sequential digits * 3) E.g.: 1234 or 4321.

Strengths (what will add to the score):

Condition	Action
Length	**Add** length * 4
Upper Case chars	**Add** Length - Number of Upper-Case Letters) *2
Lower Case chars	**Add** Length - Number of Lower-Case Letters) *2
Digits	**Add** Number of Digits * 4
Symbols	**Add** Number of Symbols * 6
Digits & Symbols in the middle	(Number of Digits or Symbols in the Middle of the Password) * 2
If (Number of Requirements Met > 3)	**Add** Number of Requirements Met * 2

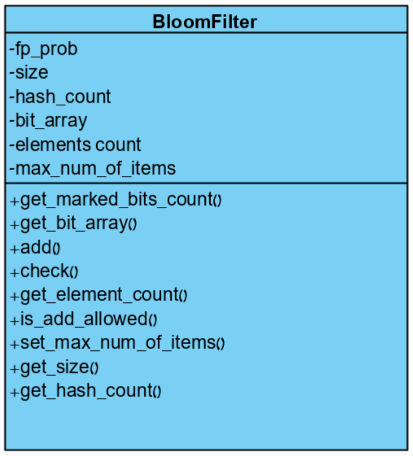
Score value legend:

```
Score == 0                  "Very Weak"
20 =< score < 40            "Weak"
40 =< score < 60            "Good"
40 =< score < 80            "Strong"
80 =< score <= 100          "Very Strong"
```

Class diagram

Our Implementation is more a python script rather than an OOP architecture. Here we present our project files with their functionality.

- **bloomfilter.py** – this class holds the implementation of the bloom filter.



- **password_strength_rules.py** – this script contains the function password_rules() which check if the given password stands within the constraints above.
- **10-million-password-list-top-1000000.txt** – a collection of the top million used passwords, downloaded from SecLists GitHub repository (https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10-million-password-list-top-1000000.txt). We use those password to prevent choosing of trivial passwords by the user.
- **gui.py** – the main script of the application. In this script we create al the visualization, which means creating the windows and integrating all the other script to produce a whole application. The main loop of the program is also within this script.

Code Hierarchy

The Project contains three Python coded files and one password text file. The Python files are located in the “src” directory and the password txt file is located in the “files” directory. The file run first is gui.py which class the bloomfilter.py file and the password_strength_rules.py file.

Installation guide

Our project was written in python on a Linux platform. There are two ways to install the dependencies and run the program:

1. manual installation:

Dependencies for ubuntu18.04:

python version 3.6 (or newer)

```
$ sudo apt-get update
$ sudo apt-get install -y python3.6
$ sudo apt-get install -y python3-pip
```

Dependencies for centos7:

```
$ sudo yum update
$ sudo yum install -y python36-setuptools
$ sudo easy_install-3.6 pip
```

Extra dependencies:

```
$ sudo pip3 install mmh3
$ sudo pip3 install bitarray
$ pip3 install pysimplegui
```

Run

To run the program, use the command:

```
python3 gui.py
```

Visualization

In wsl applications X server is required to run on the windows machine. Download can be found here: https://sourceforge.net/projects/xming/ In order to set the display to the right output use the following in the CMD:

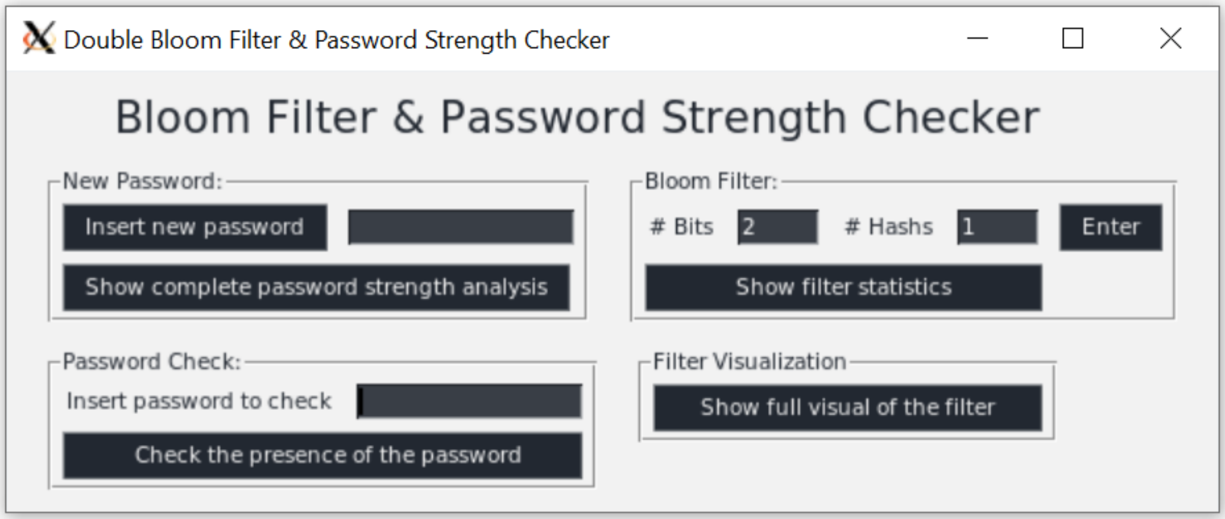
```
export DISPLAY=:0
```

2. bash script:

After running the Linux PyInstaller command, we created a bash executable file of our code, saved under the name “gui”. This executable file can be run on any machine, both virtual and not to which it its provided. In order to run the code with the given bash script:

- Enter Computer-Security-Project/src/dist/gui and use the command: ./gui.
- Watch the script display the Popup window shown below by calling all the code files in src according to their execution needs.
- Run the examples described below.

Run examples


Double Bloom Filter & Password Strength Checker
Bloom Filter & Password Strength Checker
New Password:
Insert new password
Show complete password strength analysis
Bloom Filter:
# Bits
2
# Hashs
1
Enter
Show filter statistics
Password Check:
Insert password to check
Check the presence of the password
Filter Visualization
Show full visual of the filter

Here we provide some scenarios and run examples which show how the application react in different situations. Note that the passwords bellow does not contain the “” signs.

- Set bloom filter parameters (applies to both filters):
  - Set the parameters #Bits = **2** #Hashes = **1** in the Bloom Filter section (top right) and then press Enter.
- See visualization of the bloom filter (applies to both filters):
  - Press the “Show full visual of the filter” button in the Filter Visualization (bottom right). There are binary and hexadecimal visualizations for each filter.
- Successful password insertion:
  - Enter the password “Asd@34rt” in the writing space of the New Password section (top left).
  - Press the button “Show complete password strength analysis” to see the strength analysis of the given password.
  - Press “Insert new password” to enter the password to the filter.
  - Press the “Show full visual of the filter” button in the Filter Visualization (bottom right) to see the updated visualizations for each filter.
- Password presence check:
  - Enter the password “Asd@34rt ” in the writing space of the Password Check section (bottom left) and then press the “Check the presence of the password” button. The answer should be positive.
  - Now enter the password “12345678” in the writing space of the Password Check section (bottom left) and then press the “Check the presence of the password” button. The answer should be negative.
- Unsuccessful password insertion:
  - Here we provide a sample password for each of the top requirements violation:
    - Password Length < 8: “qdh$7L”
    - Not contains uppercase letters: “q2^jms9d”
    - Not contains lowercase letters: “S24%HO)!”
    - Not contains digits: “MA*g^j(f”
    - Not contains symbols: “fW4h68md”
- False-positive scenario:
  - In order to show the FP we use above scenario (#Bits = **2** #Hashes = **1**, only one inserted password which is “Asd@34rt”. If you deviated from the steps above please reset the filters parameters to those mentioned. In this state there is only one bit lit. Now enter the password “Sad#58lb” and second bit is lit. In this point all the bits of the first filter are lit, so any additional requests will be handled by the second filter (until it will be filled too). We add another two passwords: “Vhn*72wq” and “B#7Sclq0” in order to lite all the bits of the second filter two. Now, if we check any password the answer will be positive because all the bits are lit in both filters and will match with every hash location outcome.

Conclusion

In this work we implemented a double bloom filter and a password strength checker. The implementation was done in Linux environment and written with Python. We implemented a GUI using the PySimpleGui package. Using the application, it is possible to define the filter parameters, get a visualization of the current state of the filter, insert a new password, get a full analysis of the password strength and check the presence of a given password.