Hash Cracking

52093180
CS4028 – Assessment 1
University of Aberdeen

Cracking Class

All functional elements of the project are within the Cracking Class in: `./cracking/__init__.py`.

On its creation the Class takes an array of hashes or a path to a file containing hashes. It can also take a dictionary path and a rainbow table path variable if desired. The hash file should have a hash on each line or a hash and salt separated by a "," on each line.

Password Data Class

When an instance of Cracking is initiated, a set (called passwords) is created to contain it's uncracked hashes. Each hash is structured as a Password data object before being added to the set. Password objects have values for hash, salt, password, cracked, time taken to crack, and attempts. When initiated, only hash and possibly salt will hold a value. Once a hashes password is found, cracked will = True then the password, time, and attempts value will be populated.

_crack(passwordStream):

As the logic is much the same for Tasks 1-3 I created a function that could be passed a stream of passwords that will then be iteratively hashed and compared against uncracked hashes in the passwords set. If the uncracked hashes are salted, each password in the stream will be individually salted then hashed and compared to its corresponding uncracked salted hash.

This means if passwords are salted, rather than a password being hashed once, it will be hashed as many times as there is uncracked passwords

Tasks

Task01.py '$HashPath'

```
$ python3 Task01.py './hashes/Task01Hashes.txt'
```

Running Task 01 will initiate Cracking with the provided hashes file path and call `bruteForce()`. `BruteForce()` creates a password stream `bruteForceSteam()` which yields the function `rebase(i)` with an incrementing `i` on each call. `rebase()` takes a base10 integer and converts it to the base of the provided alphabet - the default being base36 (comprising integers and lowercase characters). `bruteForceSteam()` is passed to `_crack()` which then iterates through all natural number in base36 until the input hashes are cracked. The list of cracked passwords is then output.

Task02.py & Task03.py '$HashPath' '$DictPath'

```
$ python3 Task02.py './hashes/Task02Hashes.txt' './dictionaries/PasswordDictionary.txt'
$ python3 Task03.py './hashes/Task03Hashes.txt' './dictionaries/PasswordDictionary.txt'
```

As the Password data class and `_crack()` function account for the addition (or absence) of salts, `task02.py` & `task03.py` are essentially identical in there implementation, aside from the default hashes path. Cracking is initiated with the passed hashes and dictionary file arguments, or uses the defaults if none are passed. The initialiser passes the hashes file path to `_createHashFileArray()` which creates an array

of hashes or an array of hash and salt tuples depending on the file, which is then used to create the set of Password objects.

The function `dictionaryAttack()` is then called on the object. It opens the dictionary file and creates `dictionaryStream()` which yields incremental lines of the dictionary file on each call, in effect iterating through all the passwords stored in the document. `dictionaryStream()` is passed to `_crack()` which runs until all the hashes are cracked or the dictionary runs out of passwords to try. The dictionary file is then closed and the list of cracked passwords is output.

Task04 Rainbow Tables

```
$ python3 Task04.py -h
```

```
$ python3 Task04.py \
--hashesPath="./hashes/PasswordDictionary-hashes.txt" \
--tablePath="./rainbows/PasswordDictionaryStart.rt"
```

```
$ python3 Task04.py \
--hashesPath="./hashes/PasswordDictionary-hashes.txt" \
--dictionary="./dictionaries/PasswordDictionary.txt" \
--chainLength=1000 \
--chainCount=1000 \
--strLength=8 \
--alphabet="0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

Rainbow Tables seemed like the most interesting area to further develop my cracking application. The table only holds key:value pairs, know as chains, of an end hash and a starting string which complies with a given regular expression. Each entry in the table is generated by repeatedly hashing the string and then further reducing that hash to a string of a set length and alphabet which is then hashed again, looping for a given amount of times defined by the chain length [1], [2]. We store the end of the chain hash as the key and the starting string as the value in the table.

To check if our uncracked hash is in the table, we repeatedly hash and reduce it up to the chain length, checking if it matches an end hash in the table at each iteration. If a match is found, we know that its highly likely our cracked hash is in that chain [2]. We can then regenerate the chain using the starting string, comparing hashes as we go. As there is more strings then hashes, we might not return the exact password, but this doesn't matter as it would still authenticate against the hash.

The regex of our strings as well as the amount of chains and their length vary between rainbow tables. A table with an unlimited amount of chains with length 1 would require an unlimited amount of space but no hashing and reducing making it computationally cheap to lookup hashes. On the other hand a table of 1 chain of any length would essentially be a brute force approach and incredibly computationally expensive. Rainbow Tables are a compromise of space and computation, and I'll need to find the right balance of chain length and sum.

Goals

I'll need to implement a function for generating a rainbow table and one for checking uncracked hashes again it. After the initial computation of the table, the lookup of

hashes should be relatively quick. I will need to compare the runtime of different methods to verify this.

Creation

RainbowTable Class

For consistency, I'll keep all of my code within my Cracking class. Rainbow table will initialise with an alphabet for its strings, an int indicating how long its strings should be, chain amount, and chain length. It will also be passed the parent object for its hash function, rebase function and dictionary path. A seed can be passed or will be generated based on the date. If a path to a `.rt` file is given it will be loaded, else a table will be generated and then saved to `./rainbows/*`.

Reduce()

The reduce function will take a hash and return a string that matches the alphabet and string length parameters. This is done by converting the hash to an int, rebasing it with the alphabet, reversing the string to mitigate repeats and then trimming the length.

_generate()

This function is called when a new RainbowTable object is created without an existing Rainbow Table file path to load. It populates the table with chains. For each chain the seed is multiplied by the chain number and converted to the starting string by rebasing. This is then hashed and reduced as many times as the chain is long. Both the start string and the final reduction hash is added to the table. If a dictionary file path is provided, it's used initially for the start strings.

hashLookup()

Finally, hash lookup takes a Password object with an uncracked hash value. It's hash value is iteratively looked up in the table and re-hashed as many times as there are links in the chain. If a match is found, the matches entire chain is regenerated, comparing each link with the uncracked hash. If a link's hash matches the uncracked password's then the password is set to the reduced string that created the hash and cracked is set to `True`.

rainbowAttack()

This cracker function, when called, will load a saved table or generate a new one. Every uncracked password will then be individually passed to the table's hashLookup() function.

Methods

I wanted to compare the speed and efficiency of the Rainbow attack to a standard brute force or dictionary attack. I generated a Rainbow table with 6941 chains of 10,000 length using the provided PasswordDictionary.txt for the starting strings with a base71 alphabet of:
"0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!?#*$@-_.".

I intended to create 500,000 chains, however time was a limiting factor.

I'll attempt to crack the hashes of the 100,000 top passwords "hashes/10-million-password-list-top-100000-hashes.txt"[3] with each of the 3 attacks .

```
$ python3 Task01.py './hashes/10-million-password-list-top-100000-hashes.txt'
```

```
$ python3 Task02.py './hashes/10-million-password-list-top-100000-hashes.txt'
'./dictionaries/PasswordDictionary.txt'
```

```
$ python3 Task04.py \
--hashesPath="./hashes/10-million-password-list-top-100000-hashes.txt" \
--tablePath="./rainbows/PasswordDictionaryStart.rt"
```

Conclusions

Using a dictionary attack cracked 969 of the 100 thousand in a time of 3:43, there is never a guarantee that the correct passwords for a crack will be in a dictionary file. Running the brute force attack for 3:43 cracked 18 of the passwords.

The rainbow table took ~1 hour to generate 6941 chains and created a table file of 1MB. The table theoretically holds 69,410,000 hashes. If I had been able to generate the 500,000 chains I intended, the space would have theoretically held 5,000,000,000 hashes. Of the 100,000 top passwords, it was able to crack 969 which was no improvement over the dictionary attack, the longest took 07:29:27 hours and the shortest took 00:01:05.

Due to the size of my table it was highly unlikely they any additional passwords would be found in addition to the ones used to start the chain. If space and time allowed, the generating of a several hundred GB table would vastly improve the chances of a hash collision and the subsequent cracking of the password.

What I learnt

Currently _crack generates a hash then runs through all uncracked hashes to compare for my earlier task of bruteforce and dictionary search. I would use a look up table if I was to do it again, performance drastically dropped for large sets of uncracked passwords.

Implementing a thread pool for both table generating and table lookup would also be something I'd like to work on for alleviating much of the performance issues, hashing and reducing thousands of hashes thousands of times would definitely benefit from multiple processes.

Ultimately there doesn't seem to be a take away choice for cracking. Your limitations need to be understood and then the best option can be implemented. The cost will be storage or time.

Bibliography

[1] 'Understanding Rainbow Table Attack - GeeksforGeeks', Geeks for Geeks. Accessed: Oct. 21, 2023. [Online]. Available: https://www.geeksforgeeks.org/understanding-rainbow-table-attack/

[2] 'Rainbow table - Wikipedia', Wikipedia. Accessed: Oct. 21, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Rainbow_table

[3] @danielmiessler, 'SecLists/Passwords/Common-Credentials/10-million-password-list-top-1000000.txt at master · danielmiessler/SecLists'. Accessed: Oct. 22, 2023. [Online]. Available: https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10-million-password-list-top-1000000.txt