*Hash  
Cracking*

52093180  
CS4028 – Assessment 1  
University of Aberdeen

# Cracking Class

Describe init inputs

hashesPath = None | "./hashes/Task01Hashes.txt"

Takes text file with a new “hash” or “Hash,Salt” per line. Are converted to array or hash or tuples

dictionaryPath = None | './dictionaries/PasswordDictionary.txt'

rainbowTablePath = None |

hashes = None | ['f14aae6a0e050b74e4b7b9a5b… ]

Used for testing

## Password Data Class

When an instance of Cracking is initiated with an array of hashes, a set (called passwords) is created for the class comprising the hashes structured as Password data objects. Password objects have values for hash, salt, password, cracked, 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 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.

***Explain***

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 are output.

## Task 04

1000 words describing: (1) the goals, (2) the methods, (3) the conclusions, (4) what you learned.

- [Communication] How well communicated was the work?

- [Knowledge and understanding] How much knowledge and understanding was demonstrated?

- [Skill demonstrated]How complex was what was done? How well was it done?

- [Work done] How much work was done?

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 regex compliant string 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. On the other hand a table of 1 chain of any length would require very little space for on the hash and string but would be 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.

**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 and trimming the length.

**\_generate()**

This function is called when a new RainbowTable object is created. It populates the table dictionary with chains. For each chain the seed is multiplied by the chain number and converted to the starting string. This is then 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 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.

**rainbowAttack()**

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

* Finally create a cracking function that checks if uncracked passwords are in the rainbow table and return the password if so.
* Create a function that can save and load the rainbow table
* Compare speed of dictionary attack to rainbow table

### Methods

I wanted to compare the speed and efficiency of the Rainbow attack to a standard bruteforce or dictionary attack. I generated a Rainbow table with 177,346 chains of 10,000 length using the provided PasswordDictionary.txt and as well as random generation 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” 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 './hashes/10-million-password-list-top-100000-hashes.txt' './rainbows/billions.rt'

### Conclusions

The generation took ~7 hours and created a 1MB file of 177,346 rows but theoretically the table holds 1,773,460,000 hashes. If I had been able to generate the 500,000 chains, the space would have theoretically held 5,000,000,000 hashes. Of the 100,000 top passwords, it was able to crack ∆, the longest took ∆ and the shortest took ∆. Using a dictionary attack cracked 969 of the 100 thousand in a time of 3:43, there is never a guarantee that the sought password would be known to check against. Running the brute force attack for 3:43 cracked 18 of the passwords.

Due to the size of my table there….

As the Rainbow cracked ∆, which is ∆ more than the dictionary attack, this shows that…

### What I learnt

Currently \_crack generates a hash then runs through all uncracked hashes to compare. Would use a look up table if I was to do it again, performace drasticly dropped for large uncriokded sets

Would write to run generation and lookup function so they can take advantage of multiple threads and significantly reduce there run time. I’d implement this fort all of the tasks. Looking at running on a gpu would be an interesting further development.

# Bib

[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