*Hash  
Cracking*

52093180  
CS4028 – Assessment 1  
University of Aberdeen

# Cracking Class

## 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

## Task 01

$ 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 ‘$HashPath’ ‘$DictPath’

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

Task 2 initiates Cracking with the provided hash and dictionary file, or uses the defaults if arguments aren’t passed. dictionaryAttack() is then called on the object. The function dictionaryAttack() opens the dictionary file and creates dictionaryStream() which yields incremental lines of the 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.

## Task 03

NEED TO ALLOW HASH INPUT? WORKS for T1 But not T2?

## 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. Doesn’t return the exact password, but this doesn’t matter as it would still authenticate against the hash.

### Goals

* Create rainbowTable Class
* Create a reduction function that reduces a hash to a string of a given alphabet and lenth
* Create a function that creates a rainbow table given a reduction function and a number or chains and their length
* Create a function that can save and load the rainbow table
* Finally create a cracking function that checks if uncracked passwords are in the rainbow table and return the password if so.
* Compare speed of dictionary attack to rainbow table

### Methods

### Conclusions

### What I learnt