SUTD ModularMaster in Cybersecurity
Security Tools Lab 1 Project 4
Passwords

Jefnilham Bin Jamaludin
Cyber Security Agency of Singapore

Abstract

This study was conducted as it should be of great interest to cybersecurity professionals and also the layman to have uncrackable passwords. Using multiple approaches to password generation and password quality checking, this paper aims to find out which password generation method created the set of passwords with the strongest resistance to guessability and the applicability of our password quality checker tool compared to those already available online.

1 Introduction

The flow of this paper and its methodologies follows closely to the rubrics given in the Project 4 briefing. We will elaborate on the 6 different methods used to generate passwords. Next we will elaborate on the design of the password quality checking tool and the methods used. This will be followed by evaluating of results of both password generation by statistically analyzing the character distributions and the time taken and we will also compare our tool's output with CSA's own password checking tool and Nordpass. Finally, we will discuss the strengths and the limitations and the conclusion to this paper.

2 Methods

This section will cover the 2 main implementations of this paper; password generation and password resistance to guessability. But first we will go through all the scripts and outputs so that the reader has some context.

2.1 Testing Instructions

Download all the contents in the zipped file stl1_submissions folder. Additional necessary files such as rainbow tables dependencies and rockyou.txt files has been included in this folder. Separate modules might need to be imported by another user testing out this scripts.

- Run python script `stl1_password_generator.py` that has been set to generate 5 passwords only per character combination and length for all 6 sources and will output saved csv file `stl1_all.csv`.
- Run python script `stl1_hasher.py` to output the MD5 hashes and populate this folder with the respective text files.
- Run python script `stl1_frequency.py` to output the character occurrences count in descending order and populate this folder with the respective text files.
- Run python script `stl1_tool.py` to test the password quality checker tool.

2.2 Password Generation

This subsection will encompass the methods used in password generation. In total there are a 6 methods used. Of each, all pertinent functions will be elaborated on in their respective sections. A python script `stl1_password_generator.py` ↪ creates passwords from the aforementioned 6 methods of password generation and stores in a comma-separated-value file for a total of 4,500,000 passwords. The times taken to generate per the method is also recorded as part of the script.

The script has been tested and works as planned. However, it will be far too long to only run the main script to generate such a large set of passwords. Therefore, 6 separate scripts are run and collated back.

2.2.1 Naming Convention

According to the requirements, generation must be parameterized by character classes—letters, letters and digits, letters and symbols, symbols and digits and all four classes together —and password length — 8, 12, and 20 characters long—in order to determine if these options had any effect on the randomness and ultimately guessability of generated passwords.

Therefore, in the csv file, each column will store passwords for a set of password length and character class. For our case, the first column will have passwords with length of 8 characters and the character class is only letter i.e. 8-L. To characterize according to generation method, the csv headers are also named to include that, so we will have PG_8-L representing password generation method from Passwords Generator.

The 6 password generation sources are as follows:

PG : Passwords Generator
LP : LastPass
BM : Bitmill
RJ : Random Justyy
PS : Python Secrets Module
CG : CryptGenRandom on Windows

The 15 password length and character class combinations per password generation sources as follows:

8-L, 8-LD, 8-LS, 8-DS, 8-LDS
12-L, 12-LD, 12-LS, 12-DS, 12-LDS
20-L, 20-LD, 20-LS, 20-DS, 20-LDS

2.2.2 Overall Code Structure

The main script `stl1_password_generator.py` starts with generating a csv file `stl1_all.csv`, indexing headers and the next 50,000 rows (50,000 rows because we will be generating 50,000 passwords from each password length and character class combinations per password generation sources:

```
passwords_to_generate = 50000
filename = 'stl1_all.csv'

# open csv file and index 1st column to 50k
f = open(filename, 'w')
writer = csv.writer(f)
index_list = []
i = 1
while i < (passwords_to_generate + 1):
    index_list.append(i)
    i += 1
header = ['Index']
writer.writerow(header)
for w in range(passwords_to_generate):
    writer.writerow([index_list[w]])
f.close()
```

While the methods vary, the overall implementation is roughly similar across the 6 methods with the pseudo-code of our approach as follows and will be elaborated in their respective subsections next:

```
1. create csv file indexed to 50,000

2. define main function to iterate character
    ↪ classes
   that calls function to generate passwords
      that calls function that saves to main csv

3. call main function for length 8, 12, 20
4. repeat for the 6 sources
```

As the project briefing included and suggested that we use the websites given and to find more of such similar websites, we explored more than the briefed methods to ensure complexity while keeping online sources the majority of password generation sources.

3 of the 6 password generators were sources from online websites and web-scraped with Selenium: PG, LP, BM.

1 of the 6 password generator was sourced from a making HTTP requests to application programming interface (API): RJ.

1 of the 6 password generator was sourced from Python's own Secrets module: PS.

1 of the 6 password generator was sourced from CryptGenRandom in Windows: CG, similar to Linux dev/urandom.

2.2.3 Web-scraping with Selenium from PG, LP and BM

For our case, we used Chrome browser. We initially used Firefox browser but memory leaks were evident, overusing RAM unnecessarily[1] and the time taken to generate 4,500,000 passwords would be too long. Thankfully, Chrome did not face this problem.

Naturally, each website is interfaced differently (the css elements, xpath, and object id's being referenced are differen and some websites have popups to be closed too), and as such, our Selenium script is made specifically for that version of that website on that browser.

Nonetheless, all 3 sources, our approach to web-scrape is similar. For each website, we first define a function to scrape for L, LD, LS, DS and then LDS. For each combination, we define another function to save to the previsouly created csv. With these 2 functions, we called them accordingly for password lengths of 8, 12 and finally 20. We will be using LP as an example of the 3 scripts since our approach is generally similar.

First, we define a `save_to()` function that saves the generated passwords with pandas and names headers correctly:

```
def save_to(website_shortform_x, char_x, classes,
↪   generated_password_list_x):

# use pandas to save to earlier indexed csv
    df = pd.read_csv(filename)
    df['%s_%s-%s' % (website_shortform_x, char_x,
    ↪   classes)] =
    ↪   pd.DataFrame(generated_password_list_x)
    df.to_csv(filename, index=False)
```

Secondly, we define `gen_copy_loop()` that interacts with the website, generates the passwords and saves the passwords generated into a list:

```
def gen_copy_loop(generated_password_list_x):
    passwords_generated = 0

# loop to keep generating and saving to list
    while passwords_generated <
    ↪   passwords_to_generate:
        driver.find_element_by_css_selector('div⌋
        ↪   .button:nth-child(1)').click()
        generated_password =
        ↪   driver.find_element_by_id('final_pas⌋
        ↪   s').get_attribute("value")
        generated_password_list_x.append(generat⌋
        ↪   ed_password)
        passwords_generated =
        ↪   passwords_generated + 1
```

Thirdly, main function `character_classes()` interacts with the website and iterates through the character classes in the order from L, LD, LS, DS, LDS. This main function calls `gen_copy_loop` ↪ `()` for each character class to generate passwords and finally calls `save_to()` to save to the main csv file.

```
def character_classes(char_x):
    classes = 'L'

# interact with website to set to character class
    driver.find_element_by_css_selector('#Symbol⌋
    ↪   s').click()
    driver.find_element_by_css_selector('#Number⌋
    ↪   s').click()
    driver.find_element_by_css_selector('#Nosimi⌋
    ↪   lar').click()
    generated_password_list_a = []

# call function to generate password
    gen_copy_loop(generated_password_list_a)

# call function to save to csv
    save_to(website_shortform_1, char_x,
    ↪   classes, generated_password_list_a)

# repeat for LD
# repeat for LS
# repeat for DS
# repeat for LDS
```

2.2.4 HTTP Requests to API from RJ

For RJ, we created a very simple function to create url based on the http, and a second main function to make API calls, get the response, parse it properly, save to a list and then save into the main csv, similarly with pandas.

```
# get correct http link according to bits
def RJ_merged_url(chars, b):
    url = 'https://random.justyy.workers.dev/api⌋
    ↪   /random/?cached&n=' + str(chars) + '&x='
    ↪   + str(b)
    return url

# define function
def RJ_gen(chars, b, combi):
    passwords_list = []
    i = 0

# loop to make API calls
    while i < passwords_to_generate:
        url = RJ_merged_url(chars, b)

# save and parse response
        response = requests.get(url)
        decoded_data = json.loads(response.conte⌋
        ↪   nt.decode('utf-8'))
        print(i + 1, decoded_data)
        passwords_list.append(decoded_data)
        i += 1

# save to main csv with pandas
    df = pd.read_csv(filename)
    df['RJ_%s-%s' % (chars, combi)] =
    ↪   pd.DataFrame(passwords_list)
    df.to_csv(filename, index=False)
```

2.2.5 Python Secrets Module

According to Python documentation,[2] this secrets module should be used in preference to the default pseudo-random number generator in the random module, which is designed for modelling and simulation, not security or cryptography. While both invoke the urandom module, we do not need to create a more 'secure' algorithm for generation when converting random bytes to the password format that we desire.

```
# initialize for combi variable
L = string.ascii_letters
LD = string.ascii_letters + string.digits
LS = string.ascii_letters + string.punctuation
DS = string.digits + string.punctuation
LDS = string.ascii_letters + string.digits +
↪   string.punctuation

def gen_password_secrets(length, combi,
↪   combi_string):
    passwords_list = []
    n = 0
    while n < passwords_to_generate:

#Get an instance of the secure random generator
         secrets_gen = secrets.SystemRandom()
         password =
         ↪   ''.join(secrets_gen.choice(combi)
         ↪   for i in range(length))
         n += 1
         passwords_list.append(password)

# save to main csv with pandas
         df = pd.read_csv(filename)
         df['PS_%s-%s' % (length, combi_string)]
         ↪   = pd.DataFrame(passwords_list)
         df.to_csv(filename, index=False)
```

2.2.6 Windows CryptGenRandom

Using os.urandom on a windows machine, we would be invoking CryptGenRandom, not Linux's dev/urandom.

```
# initialize for combi variable
L = string.ascii_letters
LD = string.ascii_letters + string.digits
LS = string.ascii_letters + string.punctuation
DS = string.digits + string.punctuation
LDS = string.ascii_letters + string.digits +
↪   string.punctuation

def gen_password_urandom(length, combi,
↪   combi_string):
    passwords_list = []
    n = 0

# generate seed
    while n < passwords_to_generate:
         random.seed = (os.urandom(1024))

# a simple join random combi choice
         password = ''.join(random.choice(combi)
         ↪   for i in range(length))
         n += 1
         passwords_list.append(password)

# save to main csv with pandas
         df = pd.read_csv(filename)
         df['WC_%s-%s' % (length, combi_string)]
         ↪   = pd.DataFrame(passwords_list)
         df.to_csv(filename, index=False)
```

2.3 Password Quality

Passwords are usually stored in hash instead of plaintext. For this paper, hash function used was MD5 hash as even though it is known to be cryptographically broken, it is still widely used hash function.

In python, we create a function `hasher()` to input the comma-separated value files of the password generated from all the 6 sources and output the MD5 hash in a text file.

`hasher()` first uses conditional statements to create a suffix for naming of the output of the txt file

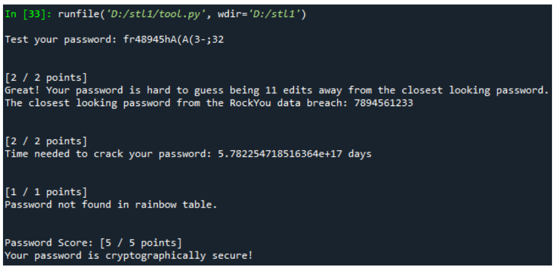
```
# take file
def hasher(plaintext_filename):
    column = 0
    for column in range(0, 15):
        column += 1

# rename string to name csv
        plaintext_csv = plaintext_filename +
        ↪   '.csv'
        if column == 1:
            suffix = '8-L'
        elif column == 2:
            suffix = '8-LD'
        elif column == 3:
            suffix = '8-LS'
        elif column == 4:
            suffix = '8-DS'
        elif column == 5:
            suffix = '8-LDS'
        elif column == 6:
            suffix = '12-L'
        elif column == 7:
            suffix = '12-LD'
        elif column == 8:
            suffix = '12-LS'
        elif column == 9:
            suffix = '12-DS'
        elif column == 10:
            suffix = '12-LDS'
        elif column == 11:
            suffix = '20-L'
        elif column == 12:
            suffix = '20-LD'
        elif column == 13:
            suffix = '20-LS'
        elif column == 14:
            suffix = '20-DS'
        elif column == 15:
            suffix = '20-LDS'
```

`hasher()` then opens an existing csv file containing the generated passwords which is generated in plaintext, runs the MD5 hash function on a cell down the same column, row by row and then iterates to the next column and so on.

```
# open csv
with open(plaintext_csv) as file:
    csvreader = csv.reader(file)
    next(csvreader)

# open txt file
    generated_md5_txt = plaintext_filename +
    ↪   '_md5_' + suffix + '.txt'
    print('Starting on %s' % generated_md5_txt)
    with open(generated_md5_txt, 'w') as f:
        for row in csvreader:
            new_row = row[column]

# run hash function and write to txt file
            new_row = [hashlib.md5(new_row.encod⌋
            ↪   e()).hexdigest()]
            for new_row_item in new_row:
                f.write('%s\n' % new_row_item)
                print(new_row_item)
```

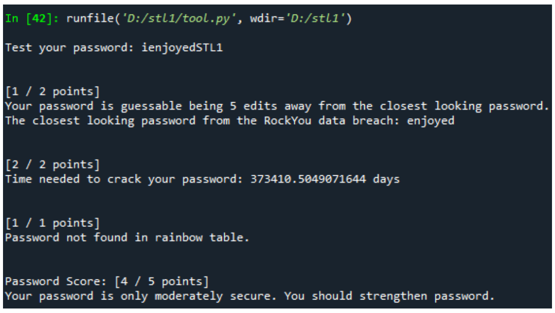
2.3.1 Overall Code Structure

The script `stl1_tool.py` takes input from the user. Next, it puts it through 3 checks to test how resistant to guesses the password is; bruteforce, Levenshtein distance and rainbow tables. For each, a score is given and accumulated to the total score.



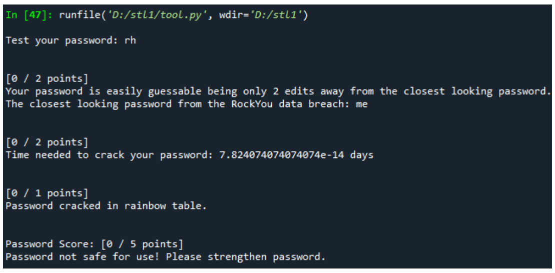
```
In [33]: runfile('D:/stl1/tool.py', wdir='D:/stl1')

Test your password: fr48945hA(A(3-;32


[2 / 2 points]
Great! Your password is hard to guess being 11 edits away from the closest looking password.
The closest looking password from the RockYou data breach: 7894561233


[2 / 2 points]
Time needed to crack your password: 5.782254718516364e+17 days


[1 / 1 points]
Password not found in rainbow table.


Password Score: [5 / 5 points]
Your password is cryptographically secure!
```

Figure 1: Strong password, score = 5



```
In [42]: runfile('D:/stl1/tool.py', wdir='D:/stl1')

Test your password: ienjoyedSTL1


[1 / 2 points]
Your password is guessable being 5 edits away from the closest looking password.
The closest looking password from the RockYou data breach: enjoyed


[2 / 2 points]
Time needed to crack your password: 373410.5049071644 days


[1 / 1 points]
Password not found in rainbow table.


Password Score: [4 / 5 points]
Your password is only moderately secure. You should strengthen password.
```

Figure 2: Moderate password, score = 3 or 4



```
In [47]: runfile('D:/stl1/tool.py', wdir='D:/stl1')

Test your password: rh


[0 / 2 points]
Your password is easily guessable being only 2 edits away from the closest looking password.
The closest looking password from the RockYou data breach: me


[0 / 2 points]
Time needed to crack your password: 7.824074074074074e-14 days


[0 / 1 points]
Password cracked in rainbow table.


Password Score: [0 / 5 points]
Password not safe for use! Please strengthen password.
```

Figure 3: Weak password, score = 0 or 1 or 2

2.3.2 Levenshtein Algorithm

The first check is by using Levenshtein distance.[3] In essence, how close the users input password is to a password in a given list of passwords. Obviously, a strong password is one that is very far away from any known leaked passwords and the converse is also true. There are 2 parts to this; first is getting the closest matched word and second is calculating how many edits are the user's input password is to that closest matched word. In our paper, the reference of known passwords are from the RockYou leak. The scoring for the Levenshtein distance is as follows:

Input passwords with less than 5 edits from a known password in RockYou password leak scores 0 / 2. Input passwords with more than or equal to 5 edits and less than or euqal to 10 edits from a known password in RockYou password leak scores 1 / 2. Input passwords with more than 10 edits from a known password in RockYou password leak scores 2 / 2.

```python
# Getting closest matched word
def closest_password(s1, s2, ratio_calc = False):
    if len(s1) > len(s2):
        s1, s2 = s2, s1
    distances = range(len(s1) + 1)
    for i2, c2 in enumerate(s2):
        distances_ = [i2+1]
        for i1, c1 in enumerate(s1):
            if c1 == c2:
                distances_.append(distances[i1])
            else:
                distances_.append(1 +
                ↪  min((distances[i1],
                ↪  distances[i1 + 1],
                ↪  distances_[-1])))
        distances = distances_
    return distances[-1]
```

Getting number of edits needed via the Levenshtein distance is complex and much readings were needed and is fairly lengthy to implement. Please find the full implementation in the attached code for alongside this paper for the function `distance()`.

2.3.3 Time to Bruteforce

The second check is the resistance to bruteforce. A strong password is one that is very resistant, possibly safe for a lifetime. This would get a score of 2 / 2. A moderately strong password can only last a year, since it is recommended for passwords to be changed regularly.[4] This would get a score of 1 / 2. A password that cannot last a year is deemed unsafe and scores 0 / 2.

The bruteforce rate is set at 100,000,000,000 passwords per second.[5] Therefore, the resistance to bruteforcing depends on the character combination and also heavily on the password length.

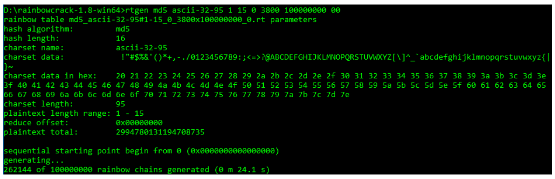
```python
# Getting total permutations
def bruteforce(string, points):
    islower = 0
    isupper = 0
    isdigit = 0
    isspecial = 0
    character_counter = 0

# if char in char class, count up
    for char in string:
        if char.islower():
            islower += 1
        elif char.isupper():
            isupper += 1
        elif char.isdigit():
            isdigit += 1
        elif not char.isalnum():
            isspecial += 1

# if counted up, can count as permutation
    if islower > 0:
        character_counter += 26
    if isupper > 0:
        character_counter += 26
    if isdigit > 0:
        character_counter += 10
    if isspecial > 0:
        character_counter += 34
```

The rest of the function calculates the time needed, in days, for bruteforcing.

2.3.4 Rainbow Tables

The third check is rainbow tables and rainbow crack has been covered in this module and is thus, trivial, while the first and second checks above are novel to this paper. Scoring as follows; if a user's input password can be cracked with a preset rainbow table, the password will score 0 / 1. So if a user's input password cannot be cracked with the same preset rainbow table, the password will score 1 / 1 for being resistant to this method.

For generation, plaintext length of 1 - 15 was chosen as this was the maximum the program can accept. All characters were to be chosen, so ascii-32-95 was chosen. Hash function is kept the same with MD5. A chain length of 10,000,000 was created as this was already 1.5 GB.



```
D:\rainbowcrack-1.8-win64>rtgen md5 ascii-32-95 1 15 0 3800 100000000 00
rainbow table md5_ascii-32-95#1-15_0_3800x100000000_0.rt parameters
hash algorithm:         md5
hash length:            16
charset name:           ascii-32-95
charset data:            !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|
}~
charset data in hex:    20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e
3f 40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f 50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f 60 61 62 63 64 65
66 67 68 69 6a 6b 6c 6d 6e 6f 70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e
charset length:         95
plaintext length range: 1 - 15
reduce offset:          0x00000000
plaintext total:        2994780131194708735

sequential starting point begin from 0 (0x0000000000000000)
generating...
262144 of 100000000 rainbow chains generated (0 m 24.1 s)
```

Figure 4: Rainbow Table generation

After generation, rainbow table was sorted and transferred to the same folder the main python script is at. This allows us to use it very easily in our program.

```python
# create function
def rcrack(string, points):

# take string convert to md5
    string_md5 =
    ↪ hashlib.md5(string.encode()).hexdigest()

# create txt file to store md5 hash
    string_md5_text = 'to_rcrack.txt'
    with open(string_md5_text, 'w') as f:
        f.write('%s' % string_md5)

# run saved rainbow table
    subprocess.run('rcrack . -l %s' %
    ↪ string_md5_text, shell=True)
    output = (subprocess.check_output('rcrack .
    ↪ -l %s' % string_md5_text,
    ↪ shell=True)).decode('utf-8')

# conditionals for scoring
    if '1 of 1' in output:
        print('[0 / 1 points]')
        print('Password cracked in rainbow
        ↪ table.')
    else:
        points.append(1)
        print('[1 / 1 points]')
        print('Password not found in rainbow
        ↪ table.')
```

3 Results

3.1 Password Generation

3.1.1 Character Distribution

While desirable, it is highly unlikely that the generation of passwords from those websites used are creating true random. It is more common for such generations to be some form of pseudo-random-number-generator (PRNG).

Since we cannot truly achieve randomness, other than using the computer to input the randomness of external environmental randomness (such as mouse movement and key presses), the next best thing is to observe if the chances of a character being chosen to form a string in the password (to be generated) are in fact 'fair'.

Similar to the probability of drawing a card from a deck being $\frac{1}{52}$, each of the characters should have the same chance of getting picked to generate a password i.e. in the case of the character class 'L' (letters only), the probability of choosing 1 letter from 52 letters (lowercase and uppercase) should also be $\frac{1}{52}$.

Another variable that is closely related to the probability of being chosen would also be the range and variance of the 50,000 list of passwords. In other words, if the password generator was fair and we list out all the characters chosen from all the 50,000 passwords for a chosen character class, the number of times the most frequently used character was chosen and the least frequently used character was chosen should be very close.

Therefore, in this section, we will be exploring the method to how we quantify the fairness the password generation is. The fairer it is, the more resistant it is to password guessability. For instance, if it were not fair, a certain character would be used more often in the password generation, as a result hackers will have an easier time bruteforcing with that knowledge.

To find out, the function `frequency_checker()` was created to input the comma-separated value files of the password generated from all the 6 sources and output the count and rank of the characters.

`frequency_checker()` first uses conditional statements to create a suffix for naming of the output of the txt file, similar to `hasher()`

The main difference is that `frequency_checker()` in the main script that initiates a character list to store and restore at a new loop which is a new column. The program goes down the same column, row by row and then iterates to the next column and so on.

```python
character_list = []

# open csv of generated passwords
with open(plaintext_csv) as file:
    csvreader = csv.reader(file)
    next(csvreader)

# for renaming txt file
    generated_frequency_txt = plaintext_filename
    ↪ + '_frequency_' + suffix + '.txt'
    with open(generated_frequency_txt, 'w') as f:
        for row in csvreader:
            new_row = row[column]
            for new_row_item in new_row:
                character_list.append(new_row_it⌋
                ↪ em)

# to display in decreasing order as a dict
        counter_string = Counter(character_list)
        for key, value in
        ↪ counter_string.most_common():
            key_value_pair = str(key) + ':' +
            ↪ str(value)
            f.write('%s\n' % key_value_pair)
```

So if we run the program `frequency_checker('stl1_rj')`, the output is the text file and the contents as follows, ranked from top to bottom, for example, for `stl1_rj_frequency_8-DS.txt` is as follows:

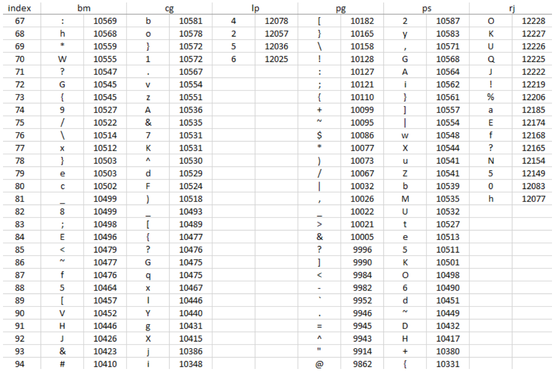
```
# in decreasing order, displaying here in this
↪ paper only the first, second, second-last and
↪ last key-value pair
~:14008
[:13999
.
.
.
{:13558
2:13534
```

Therefore, for `stl1_rj_frequency_8-DS.txt`, we can see that the most frequently chosen character for is `~` with 14008 occurrences while the least frequently chosen character is `2` with 13534 occurrences.

However, in the interest of time, we will be only choosing the largest dataset per password generator. This means we will choose all 6 sets of character combinations 'LDS' and 20 characters for analysis of character distribution:

```
stl1_pg_frequency_20-LDS.txt
stl1_lp_frequency_20-LDS.txt
stl1_bm_frequency_20-LDS.txt
stl1_rj_frequency_20-LDS.txt
stl1_ps_frequency_20-LDS.txt
stl1_cg_frequency_20-LDS.txt
```

Porting the content over to an excel spreadsheet for analysis, listing the generation sources as column headers as follows:



index	bm		cg		lp		pg		ps		rj	
67	:	10569	b	10581	4	12078	[	10182	2	10587	O	12228
68	h	10568	o	10578	2	12057	}	10165	y	10583	K	12227
69	*	10559	}	10572	5	12036	\	10158	,	10571	U	12226
70	W	10555	1	10572	6	12025	!	10128	G	10568	Q	12225
71	?	10547	.	10567			:	10127	A	10564	J	12222
72	G	10545	v	10554			;	10121	i	10562	!	12219
73	{	10545	z	10551			{	10110	}	10561	%	12206
74	9	10527	A	10536			+	10099	]	10557	a	12185
75	/	10522	&	10535			~	10095	|	10554	E	12174
76	\	10514	7	10531			$	10086	w	10548	f	12168
77	x	10512	K	10531			*	10077	X	10544	?	12165
78	}	10503	^	10530			)	10073	u	10541	N	12154
79	e	10503	d	10529			/	10067	Z	10541	5	12149
80	c	10502	F	10524			|	10032	b	10539	0	12083
81	_	10499	)	10518			,	10026	M	10535	h	12077
82	8	10499	_	10493			_	10022	U	10532		
83	;	10498	[	10489			>	10021	t	10527		
84	E	10496	{	10477			&	10005	e	10513		
85	<	10479	?	10476			?	9996	5	10511		
86	~	10477	G	10475			]	9990	K	10501		
87	f	10476	q	10475			<	9984	O	10498		
88	5	10464	x	10467			-	9982	6	10490		
89	[	10457	l	10446			`	9952	d	10451		
90	V	10452	Y	10440			.	9946	~	10449		
91	H	10446	g	10431			=	9945	D	10432		
92	J	10426	X	10415			^	9943	H	10417		
93	&	10423	j	10386			"	9914	+	10380		
94	#	10410	i	10348			@	9862	{	10331		

Figure 5: Indexed character count per source

We can now see that password generated from LP and RJ for all character classes LDS actually in fact do not use all the 94 characters. LP uses only 70 characters while RJ uses 81 characters when they should be using all 94. The implications of this is that the overall strength and resistance to bruteforce is reduced. For instance, the total permutations possible from BM, CG, PG and PS are 94^{20} while RJ only has 81^{20} and even worse, LP has only 70^{20}. Therefore, we will be excluding RJ and LP from further analysis.

Next we plot out the overall trend to see the fairness of choosing a character i.e. the distribution of the character frequencies:

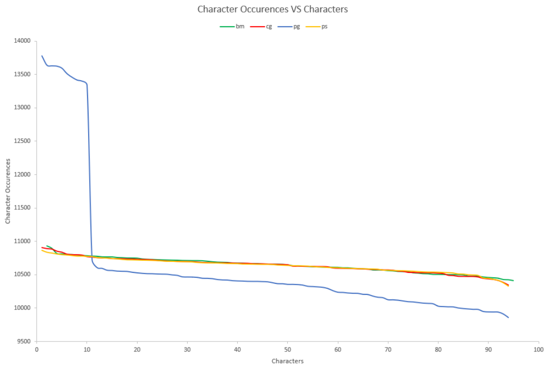
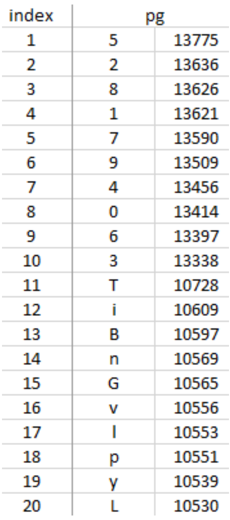


Figure 6: Distribution of character frequencies from rank 1st (left-end) to 94th (right-end) character

In the figure above, each line represents the password generation source. Blue = PG, Green = BM, Red = CG, Yellow = PS. It should be clear that the blue line (PG) is not a fair distribution as the algorithm it uses to generate secure passwords creates favours certain characters much more than the rest up till around character number 10. Beyond character number 10, the rest of the characters can be seen to have an increasingly lower chance of being chosen at a greater rate (steeper downwards gradient) than that of the other 3 password generators.



index	pg	
1	5	13775
2	2	13636
3	8	13626
4	1	13621
5	7	13590
6	9	13509
7	4	13456
8	0	13414
9	6	13397
10	3	13338
11	T	10728
12	i	10609
13	B	10597
14	n	10569
15	G	10565
16	v	10556
17	I	10553
18	p	10551
19	y	10539
20	L	10530

Figure 7: Top 20 most frequent character occurrences of PG

So upon checking PG's character frequency distribution on its own, we can see that the top 10 most chosen characters are digits only, being chosen averaging at over 13k times compared to the rest of the 84 characters (94 total characters minus 10 digit characters) averaging at only 10k times. This suggests that the algorithm used in PG's password generation mechanism is not fair and highly biased towards digits and thus we will exclude PG for further analysis. So we move on to the remaining 3 password generators; BM, CG, PS.

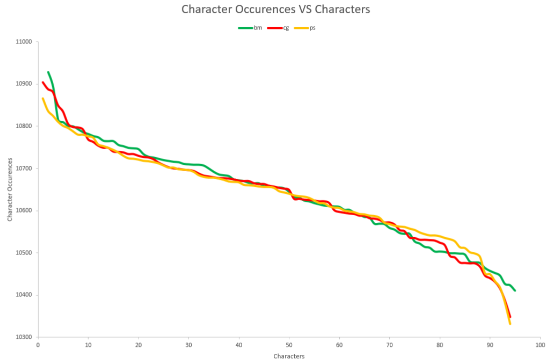
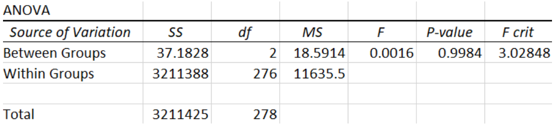


Figure 8: Distribution of character frequencies from rank 1st (left-end) to 94th (right-end) character

From this zoomed in graph that only includes BM, CG and PS, we can see that the generated passwords are very similar in trend. This similarity suggests that the generation methods of this sources are the similar. Therefore, we need to use statistical methods to distinguish these 3 sets of sources, in this case, ANOVA.



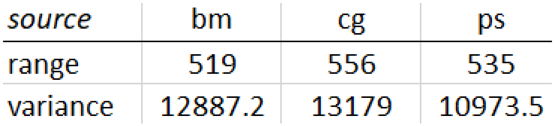
ANOVA

Source of Variation	SS	df	MS	F	P-value	F crit
Between Groups	37.1828	2	18.5914	0.0016	0.9984	3.02848
Within Groups	3211388	276	11635.5			
Total	3211425	278				

Figure 9: ANOVA results

Therefore, the null hypothesis in ANOVA is that there is no difference in means of the 3 groups is not rejected as the p-value = 0.9984 is over 0.05. This large p-value also suggest the strong similarity between the 3 sources.

For further analysis, we calculate the range to see how far apart the most used character is from the least used character for all 3 sources. The more further apart they are, the more the disparity there is in the character getting chosen for password generation. We also used variance to see how far from the mean and how far from each other the characters are. The smaller the variance, the more identical the chances of each characters are in being chosen and thus, the more fair the generation method is. The more fair it is, the less guessable the password is assumed to be.



source	bm	cg	ps
range	519	556	535
variance	12887.2	13179	10973.5

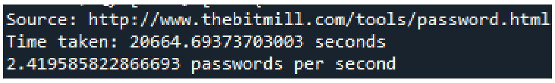
Figure 10: Range and variance of BM, CG, PS

Between these 3 groups, PS has a range lower than CG and higher than BM and PS has the smallest variance. These suggests that within the very similar 3 sets of password, PS is possibly the best choice among the 6 sources. PS is followed by BM that has the smallest range but a much larger variance, while still being lower than that of CG's.

This results is good and can be deemed a successful experiment already as it has proven that the Python Secrets module, since its conception was intended for cryptography, trumps all the other 5 generation sources. Also, since the PS, BM, and CG are statistically very similar, it suggests that our manually created algorithm for CG is a strong algorithm that matches PS's results. Additionally, BM is the next best choice next to PS as it creates a random seed based on the computer's clock.

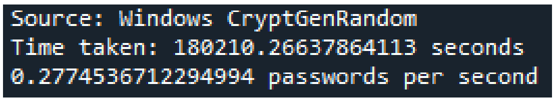
3.1.2 Time to generate

We recorded the time to generate all the 4,500,000 passwords. i.e. 750,000 passwords per source.



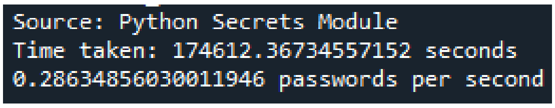
```
Source: http://www.thebitmill.com/tools/password.html
Time taken: 20664.69373703003 seconds
2.419585822866693 passwords per second
```

Figure 11: Time taken for BM



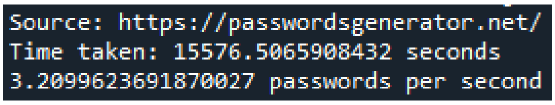
```
Source: Windows CryptGenRandom
Time taken: 180210.26637864113 seconds
0.2774536712294994 passwords per second
```

Figure 12: Time taken for CG



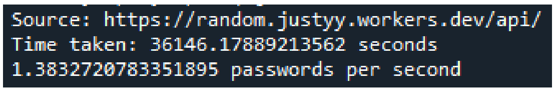
```
Source: Python Secrets Module
Time taken: 174612.36734557152 seconds
0.28634856030011946 passwords per second
```

Figure 13: Time taken for PS



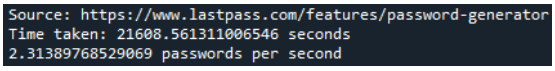
```
Source: https://passwordsgenerator.net/
Time taken: 15576.5065908432 seconds
3.2099623691870027 passwords per second
```

Figure 14: Time taken for PG



```
Source: https://random.justyy.workers.dev/api/
Time taken: 36146.17889213562 seconds
1.3832720783351895 passwords per second
```

Figure 15: Time taken for RJ

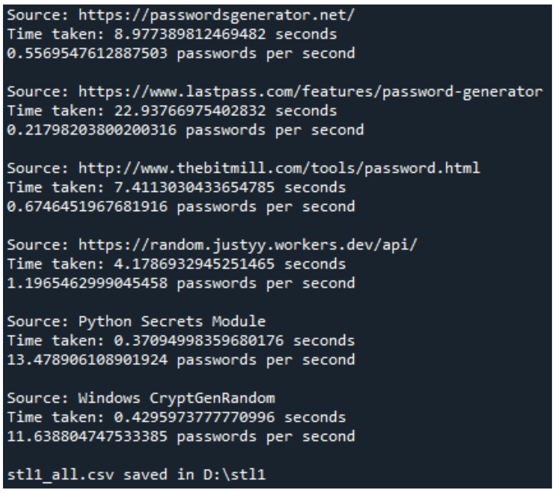


```
Source: https://www.lastpass.com/features/password-generator
Time taken: 21608.561311006546 seconds
2.31389768529069 passwords per second
```

Figure 16: Time taken for LP

The total time taken shows fastest results from Selenium based work, followed by http request approach and lastly by Python Secrets and Windows CryptGen. This shows that for large generation of passwords, Selenium still holds the top spot for automated testing. However, it was observed that for PS and CG, the time taken to generate a password starts to increase with the time the script has been running. This is why the resulting total time taken is drastically slowed in the order of 10 times longer taken for PS and CG compared to the other 4 methods.

To demonstrate, we run all the password generators in the same main script and only set to generate 5 passwords:



```
Source: https://passwordsgenerator.net/
Time taken: 8.977389812469482 seconds
0.5569547612887503 passwords per second

Source: https://www.lastpass.com/features/password-generator
Time taken: 22.93766975402832 seconds
0.21798203800200316 passwords per second

Source: http://www.thebitmill.com/tools/password.html
Time taken: 7.4113030433654785 seconds
0.6746451967681916 passwords per second

Source: https://random.justyy.workers.dev/api/
Time taken: 4.1786932945251465 seconds
1.1965462999045458 passwords per second

Source: Python Secrets Module
Time taken: 0.37094998359680176 seconds
13.478906108901924 passwords per second

Source: Windows CryptGenRandom
Time taken: 0.42959737777770996 seconds
11.638804747533385 passwords per second

stl1_all.csv saved in D:\stl1
```

Figure 17: Time taken for all 6 sources

Therefore, as previously mentioned, PS and CG actually creates passwords at an extremely fast rate. However, when using for generating large amounts of passwords, it can be seen that the time it takes drastically reduces, unlike the other 4 sources from the web that continues to generate passwords at an approximately constant rate.

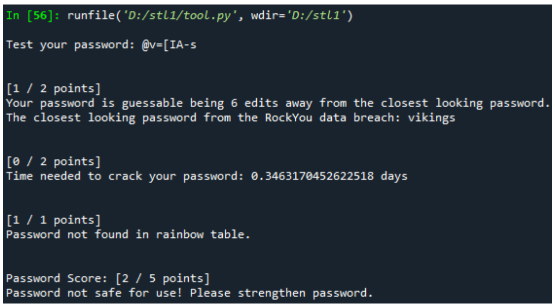
3.2 Password Quality

3.2.1 Tool Output

With passwords generated, and tool to check the passwords' resistance to guessability, this section will evaluate tool's results and compare with other tools online. We randomly picked a password source, BM, and from its csv, we chose the password string in the first row for the headers 8-LDS, 12-LDS and 20-LDS. The 3 passwords chosen are as follows:

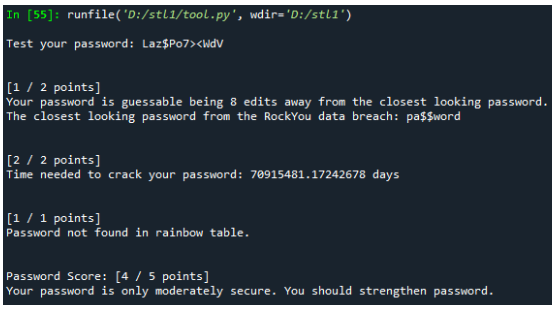
```
8-LDS = @v=[IA-s
12-LDS = Laz$Po7><WdV
20-LDS = &B-_~$>=7@E<h<<ga~cO
```

Our tool correctly certifies the 3 different passwords as what we expect, the 8-LDS being weak, 12-LDS being moderate, and 20-LDS being cryptographically strong. The images below are screenshots of the output of the `stl1_tool.py` being used in Spyder IDE.



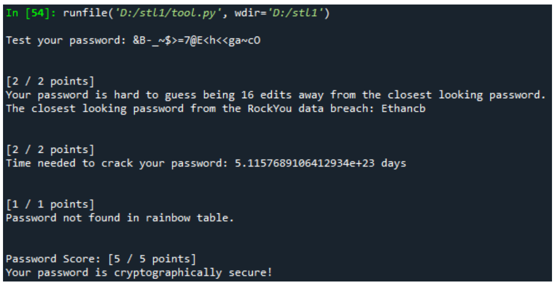
```
In [56]: runfile('D:/stl1/tool.py', wdir='D:/stl1')

Test your password: @v=[IA-s


[1 / 2 points]
Your password is guessable being 6 edits away from the closest looking password.
The closest looking password from the RockYou data breach: vikings


[0 / 2 points]
Time needed to crack your password: 0.3463170452622518 days


[1 / 1 points]
Password not found in rainbow table.


Password Score: [2 / 5 points]
Password not safe for use! Please strengthen password.
```

Figure 18: `@v=[IA-s` = weak



```
In [55]: runfile('D:/stl1/tool.py', wdir='D:/stl1')

Test your password: Laz$Po7><WdV


[1 / 2 points]
Your password is guessable being 8 edits away from the closest looking password.
The closest looking password from the RockYou data breach: pa$$word


[2 / 2 points]
Time needed to crack your password: 70915481.17242678 days


[1 / 1 points]
Password not found in rainbow table.


Password Score: [4 / 5 points]
Your password is only moderately secure. You should strengthen password.
```

Figure 19: `Laz$Po7><WdV` = moderate

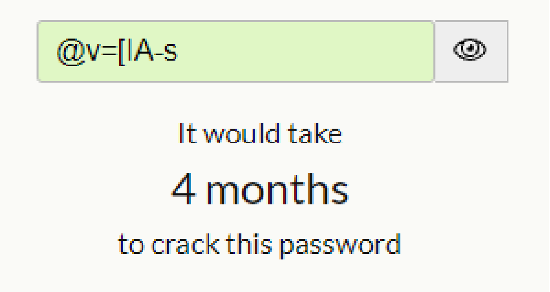


```
In [54]: runfile('D:/stl1/tool.py', wdir='D:/stl1')

Test your password: &B-_~$>=7@E<h<<ga~cO


[2 / 2 points]
Your password is hard to guess being 16 edits away from the closest looking password.
The closest looking password from the RockYou data breach: Ethancb


[2 / 2 points]
Time needed to crack your password: 5.1157689106412934e+23 days


[1 / 1 points]
Password not found in rainbow table.


Password Score: [5 / 5 points]
Your password is cryptographically secure!
```

Figure 20: `&B-_~$>=7@E<h<<ga~cO` = strong

To evaluate the accuracy of our own developed tool, we test the same set of 3 chosen passwords on 2 other online password strength checker.

Check 1: CyberSecurity Agency of Singapore's own password checking website.[6] CSA's password checker has 4 outcomes, seen from the colour of the text input field, ranked from strongest to weakest: Green, light green, orange, red.



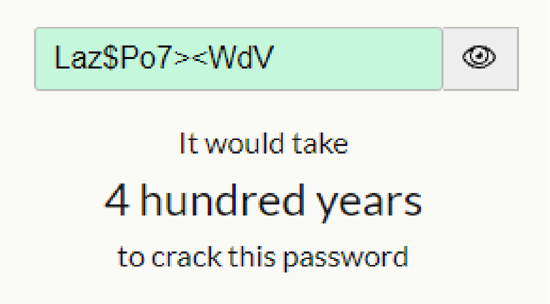
@v=[IA-s

It would take

4 months

to crack this password

Figure 21: `@v=[IA-s` = 4 months



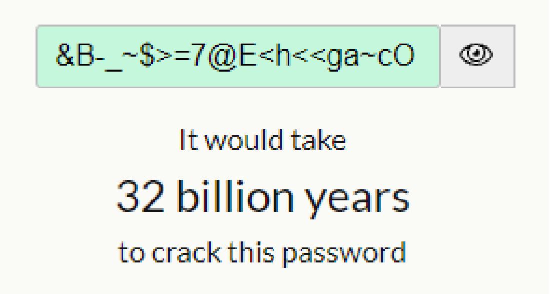
Laz$Po7><WdV

It would take

4 hundred years

to crack this password

Figure 22: `Laz$Po7><WdV` = 400 years



&B-_~$>=7@E<h<<ga~cO

It would take

32 billion years

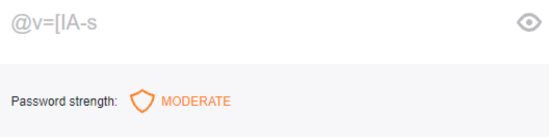
to crack this password

Figure 23: `&B-_~$>=7@E<h<<ga~cO` = 32 B years

Comparing our tool with CSA's tool, we can see that our tool's requirements are more stringent than CSA's. While the 20-LDS password performed the same and ended up in the highest rank of safety (green for CSA and 5 / 5 for our tool), the 12-LDS and 8-LDS had different outcomes. On my tool, 12-LDS was categorized as moderate with score 4 / 5 but on CSA's checker, it is categorized as strong (green colour i.e. highest security ranking). Similarly, on my tool, 8-LDS is categorized as weak with score of 2 / 5 only but on CSA's website, it is the 2nd highest category of safety (light-green).

Therefore, my own tool is more stringent in creating a cryptographically secure password than CSA's tool. Interestingly, CSA's password strength evaluation algorithm is based on zxcvbn by Daniel Lowe Wheeler.

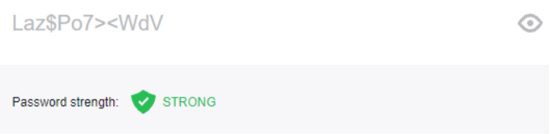
Check 2: Nordpass's online password checker.[7] Nordpass is a popular password manager launched in 2019 and the team also earlier developed NordVPN. Nordpass has 3 outcomes, 'strong', 'moderate' and 'weak', similar to my own tool.



@v=[IA-s

Password strength: MODERATE

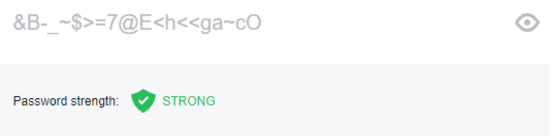
Figure 24: `@v=[IA-s` = moderate



Laz$Po7><WdV

Password strength: STRONG

Figure 25: `Laz$Po7><WdV` = strong



&B-_~$>=7@E<h<<ga~cO

Password strength: STRONG

Figure 26: `&B-_~$>=7@E<h<<ga~cO` = strong

Comparing our tool with Nordpass, a similar observation is made with that of comparing my tool with CSA's tool. Using Nordpass, it classifies the 8-LDS password as moderate and the 12-LDS password as strong, while my tool classifies the 8-LDS password as only weak and the 12-LDS password as only moderate.

Therefore, my own tool is more stringent in creating a cryptographically secure password than Nordpass' tool.

4 Discussion

4.1 Strengths

As the complexity of the work itself holds a significant part of the marking rubrics, this section will elaborate on how complexity in our approach is ensured.

4.1.1 Implementations Complexity

For password generation, our approach ensured complexity of methods was utilized. Although the project 4 brief intended for us to use only online password managers for all 6 sources and at most try to script 1 using the urandom module, our approach did much more than the aforementioned. Method 1,2,3 used Selenium mainly. Method 4 used HTTP requests. Method 5 used Python's new secrets module. Method 6 used the urandom module on Windows which was cryptgenrandom. We also added extra utilities such as time taken to add on another layer of evaluation.

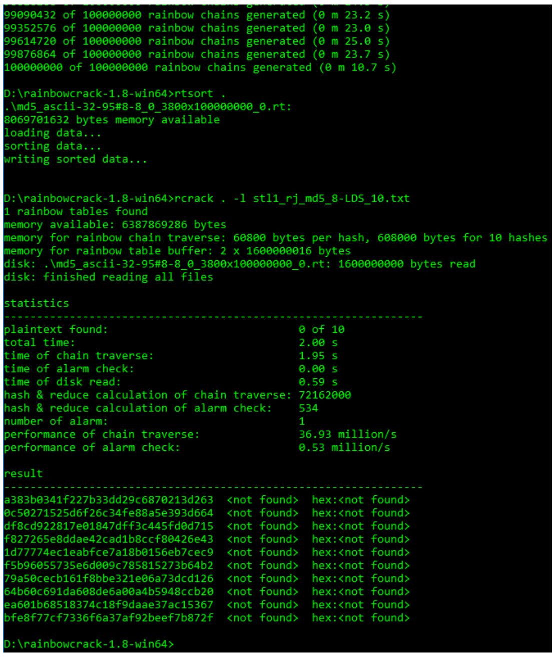
For password quality checker tool, our approach was mcuh more novel than requested, as in the project 4 brief, it was mentioned to create our own tool based on the various mechanisms we learned in Module 2 which was burteforce, rainbow cracks, hashcat. However, we introduced the Levenshtein's algorithm for distance and ratio. This adds a new level of realistic password guessing on top of the other main usual password crackers that we used (rainbow tables and time to bruteforce). In addition to this tool, a statistical analysis was made on the huge set of password generated to determine which of the 6 sources is the best one in generating cryptographically secure password.

4.2 Limitations

Both hardware and time are the main limitations that go hand-in-hand in preventing further deep-diving of this paper.

4.2.1 Time

For the password generation evaluation, we wanted to use both rainbow tables and hashcat to evaluate how many of the total passwords we can crack. Therefore, we created a script in Section 2.2 to create MD5 hashes. With that, we can get a success rate from each of the 6 password generation sources and can make a stronger conclusion on the best generator using the MD5 hashes on rainbow tables and hashcat. However, the dataset was too large and was taking too long so we tried to test smaller sets of 10 passwords but would always crack zero, rendering this method useless.



```
99090432 of 100000000 rainbow chains generated (0 m 23.2 s)
99352576 of 100000000 rainbow chains generated (0 m 23.0 s)
99614720 of 100000000 rainbow chains generated (0 m 25.0 s)
99876864 of 100000000 rainbow chains generated (0 m 23.7 s)
100000000 of 100000000 rainbow chains generated (0 m 10.7 s)

D:\rainbowcrack-1.8-win64>rtsort .
.\md5_ascii-32-95#8-8_0_3800x100000000_0.rt:
8069701632 bytes memory available
loading data...
sorting data...
writing sorted data...


D:\rainbowcrack-1.8-win64>rcrack . -l stl1_rj_md5_8-LDS_10.txt
1 rainbow tables found
memory available: 6387869286 bytes
memory for rainbow chain traverse: 60800 bytes per hash, 608000 bytes for 10 hashes
memory for rainbow table buffer: 2 x 1600000016 bytes
disk: .\md5_ascii-32-95#8-8_0_3800x100000000_0.rt: 1600000000 bytes read
disk: finished reading all files

statistics
---------------------------------------------------------------
plaintext found:                             0 of 10
total time:                                  2.00 s
time of chain traverse:                      1.95 s
time of alarm check:                         0.00 s
time of disk read:                           0.59 s
hash & reduce calculation of chain traverse: 72162000
hash & reduce calculation of alarm check:    534
number of alarm:                             1
performance of chain traverse:               36.93 million/s
performance of alarm check:                  0.53 million/s

result
---------------------------------------------------------------
a383b0341f227b33dd29c6870213d263  <not found>  hex:<not found>
0c50271525d6f26c34fe88a5e393d664  <not found>  hex:<not found>
df8cd922817e01847dff3c445fd0d715  <not found>  hex:<not found>
f827265e8ddae42cad1b8ccf80426e43  <not found>  hex:<not found>
1d77774ec1eabfce7a18b0156eb7cec9  <not found>  hex:<not found>
f5b96055735e6d009c785815273b64b2  <not found>  hex:<not found>
79a50cecb161f8bbe321e06a73dcd126  <not found>  hex:<not found>
64b60c691da608de6a00a4b5948ccb20  <not found>  hex:<not found>
ea601b68518374c18f9daae37ac15367  <not found>  hex:<not found>
bfe8f77cf7336f6a37af92beef7b872f  <not found>  hex:<not found>

D:\rainbowcrack-1.8-win64>
```

Figure 27: Minimizing dataset for rainbow table but not being able to crack any passwords

For the password quality checker tool implementation, we also wanted to add in more tests such as using rainbow tables with larger chain length. However, we are limited by time in this regard. We also wanted to use the larger rockyou text file (136 MB) that we had but it will take too long for the tool to output results so had to use a 6 MB file.

4.2.2 Hardware

All programs run and elaborated on in this paper was mainly done on an Acer Swift 3x (released 2020) and does not have dedicated GPU, instead it has integrated GPU with CPU. This created a problem that was not faced on the supporting machine ASUS K401U (purchased 2016), however had too much of a drop in computational power overall. Implementations such as hashcat was not feasible. Therefore hardware and time was the main constraint to this project and can be further explored for a more holistic tool.

5 Conclusion

Evaluating character frequencies and time to generate, we conclude that among all 6 password generation sources, the Python Secrets module is the most cryptographically secure and this is to be expected as this module was created intended to be used for cryptography-purposes. Although generating large amounts of passwords with this module is time consuming, if we were to only generate a handful of passwords, the Python Secrets module is the fastest generator.

We also conclude that our password quality tool checker utilizing the Levenshtein algorithm, time to bruteforce and rainbow tables is more stringent than some other online sources in classifying a password as strong, making it more security-focused than user-focused.

References

[1] Selenium using too much ram with firefox. https://stackoverflow.com/questions/55072731/selenium-using-too-much-ram-with-firefox. Last Accessed: 2021-09-20.

[2] secrets — generate secure random numbers for managing secrets. https://docs.python.org/3/library/secrets.html. Last Accessed: 2021-09-20.

[3] Fuzzy string matching in python. https://www.datacamp.com/community/tutorials/fuzzy-string-python. Last Accessed: 2021-09-20.

[4] How often should you change your passwords? https://www.keepersecurity.com/blog/2020/12/21/how-often-should-you-change-your-password-keeper-security/#:~:text=Most%20tech%20professionals%20recommend%20your,password%20is%20to%20begin%20with. Last Accessed: 2021-09-20.

[5] A computer can guess more than 100,000,000,000 passwords per second. still think yours is secure? https://gcn.com/articles/2020/09/15/password-security.aspx. Last Accessed: 2021-09-20.

[6] Password checker by gosafeonline. https://www.csa.gov.sg/gosafeonline/Resources/Password-Checker. Last Accessed: 2021-09-20.

[7] How secure is my password? https://nordpass.com/secure-password/. Last Accessed: 2021-09-20.