Password Pattern Analysis and Policy-Based Mask Generation For Behavioral Password-Choosing Trends

Hirsh Guha

May 16, 2021

1 Introduction & Relevance

Large scale corporate data-leaks happen almost weekly. Often, the most worrying leaked piece of information are passwords. Through these leaks and penetration testing, all over the internet are lists of the top hundred, thousand, or even ten million most common passwords. The price of these leaks is certainly nontrivial, if companies like Cloudflare, Github, Google, etc. were to have large-scale data breaches, this could easily cost the company billions of dollars, as well as a large hit to their public perception. Subsequently, penetration testing and security analysis companies make up a multi-million dollar industry, and the study of password security and password choosing behaviours continues to be an important topic for study from various and novel perspectives.

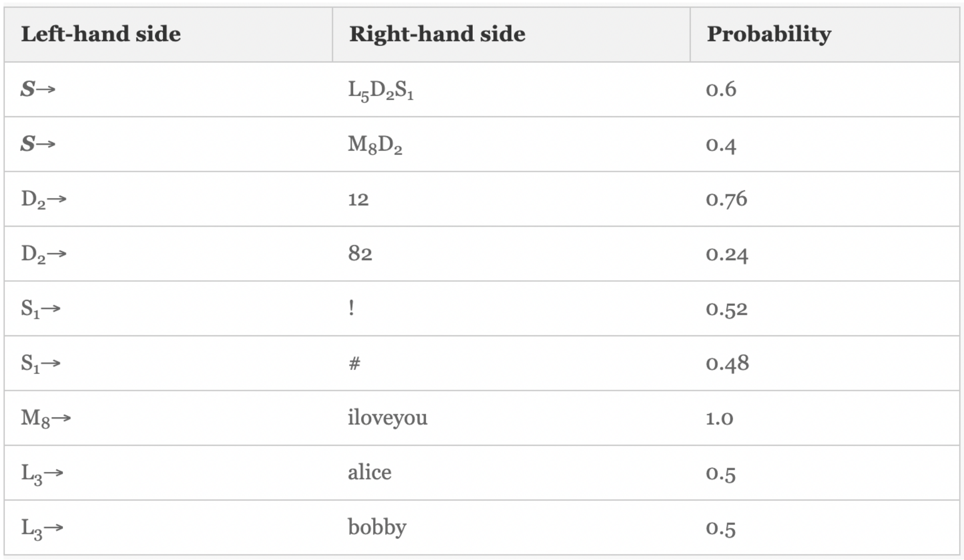
Various methods from brute force to social engineering to rainbow tables are used with various success by malicious actors. Almost 100% of our online data is secured with little more than an alphanumeric combination that can be potentially guessed, and result in huge damage to social or financial standings. For instance, if one were to know your bank password, or have access to a social media account.

Here, I will provide a methodology for analysis of these passwords gives us insight into the psychology of how humans choose passwords, as well as what password cracking techniques would be most effective as well as a methodology for policy-based mask generation for improving password cracking methods. I hope to answer and explore the following key questions and topics:

1. From a linguistics perspective, what is the makeup of a password?
2. What patterns can in password choosing behaviour can be extracted and how can those patterns be used to improve password cracking abilities?
3. How could passwords be masked(tagged) in a way that reduces computational load and time?

2 Literature Survey & Related Work

Password cracking and analysis have been studied time and time again as new methods and further refinements narrow the discussion of what makes passwords effective or ineffective. For a general overview on the methods of password cracking, I look to Simon Marechal's paper on advancements



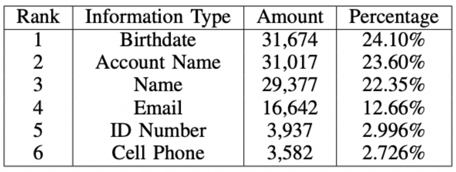
Left-hand side	Right-hand side	Probability
S→	$L_5D_2S_1$	0.6
S→	M_8D_2	0.4
D_2→	12	0.76
D_2→	82	0.24
S_1→	!	0.52
S_1→	#	0.48
M_8→	iloveyou	1.0
L_3→	alice	0.5
L_3→	bobby	0.5

Figure 1: An example of a probabilistic context free grammar that could be generated from a password list

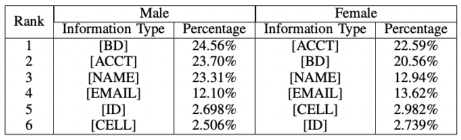
in password cracking, where he discusses in general the various methods taken, including rainbow tables, , Marcov chain tools, open ciphers, and others. He discusses the trade-offs algorithmically between time, space, and generative power necessary [8]. We will look at 4 areas of password analysis and cracking:

Probalistic Context Free Gramars The research of password strength and meta-analysis on cracked password is a common area of study. One of the most referenced papers is that by Matt Weir et al. He offers his take on using probabilistic context-free grammars to generate word-mangling rules for password guessing through a trained model [12]. For reference of how he a probabilistic context free grammar for passwords would be represented, see Figure 1, a table from Sudhir Aggarwal, Shiva Houshmand and Matt Weir on new technologies in password cracking[1]. This approach proved to be more effective than dictionary attacks, cracking somewhere between 28%-129% more passwords than Jack The Ripper, a popular penetration testing tool. This work is further expounded upon by Sudhir Aggarwal, which further refined the CFG's through the use of personal information that could be aquired via social engineering means, as is discussed below [5].

Leet & Password Mangling Li et al. notes precisely what I did in my hypothesis, that a majority of the most common passwords are basic English words, or some combination of basic English words [6]. Because this is generally mainstream knowledge nowadays, a common technique used to enhance password strength is to use ”Leet”, which is a method to replace common letters with numbers, such as replacing an ”E” in a password with a ”3”, or an ”I” with a 1, so that the word ”intrepid” would become ”1ntr3p1d”, which might be harder to guess via a dictionary attack. However, Li notes that ”only 5% of high-frequency Leet replacement could increase the cracking rate by 0.55%” [6]. As I will show later, it is fairly easy to account for Leet in a dictionary attack, at the cost of time. Along a similar vein would be to replace the English letter with a corresponding greek



Rank	Information Type	Amount	Percentage
1	Birthdate	31,674	24.10%
2	Account Name	31,017	23.60%
3	Name	29,377	22.35%
4	Email	16,642	12.66%
5	ID Number	3,937	2.996%
6	Cell Phone	3,582	2.726%



Rank	Male		Female	
	Information Type	Percentage	Information Type	Percentage
1	[BD]	24.56%	[ACCT]	22.59%
2	[ACCT]	23.70%	[BD]	20.56%
3	[NAME]	23.31%	[NAME]	12.94%
4	[EMAIL]	12.10%	[EMAIL]	13.62%
5	[ID]	2.698%	[CELL]	2.982%
6	[CELL]	2.506%	[ID]	2.739%

Figure 2: Frequency distributions from Wang et al. on the use of personal infromation such as dates and account names in passwords on the left, and broken down by gender on the right

letter or another alphabet, though this might be far less common as greek letters are not available on American-style keyboards, and would have to be pasted from a clipboard.

Social Engineering Medlin et al. goes over the implications of social engineering in obtaining passwords. Social engineering is the art of manipulating people so they give up confidential information, such as in phishing attacks. In her study, Medlin simulate a social engineering attack in five different hospitals of varying sizes with the goal of obtaining employees passwords with techniques such as pretending to be from the IT department, and requesting that the employee give their password for some purpose [9]. 73% of respondents shared their password. Even in the situation where employees are properly trained to never give out their password, social engineering methods could be used to obtain other identifying information, such as birthday's, family members names, pets names, or other important dates and events in a person's life. Li, Wang et al. studied the use of personal information in passwords[7], and her results are shown in Figure 2.

Zipf's Law Ding Wang et al. published a fascinating paper showing that password distrubutions follow Zipf's law, which states that the frequency of any word is inversely proportional to its rank in the frequency table. So word number n has a frequency proportional to 1/n[11]. More precisely,

$$f_r = Cr^{-s}$$

where the value of s that best fit the data depended on the set of passwords, but estimated between 0.46 to 0.91. John Cook generalizes these results further, showing that you can bound the sum in the denominator from above and below with integrals, as in the integral test for series convergence. This gives us a way to see more easily how C depends on its parameters[2].

$$1 + \int_1^N x^{-s} dx > \frac{1}{C} = \sum_{r=1}^{N} r^{-s} > 1 + \int_2^{N+1} x^{-r} dx$$

which he simplifies to

$$1 + \frac{N^{1-s} - 1}{1-s} > \frac{1}{C} > 1 + \frac{(N+1)^{1-s} - s^(1-s)}{1-s}$$

To better understand this, let's look at an example. Suppose you have N = 1,000,000, the number of passwords. The range of s values found by Wang et al varied from roughly 0.5 to 0.9, so we can look at both ends of the bounds. If $s = 0.5$, C is roughly 0.0005, and the most common password appears about 500 times. Set s to 0.9, C to roughly 0.033, and the most common password appears about 33,000 times[2]. While this may seem like a large bound, the margin of error is actually less than 0.005, which might show that Wang et al's estimations are quite tiightly bound.

3 Data

One of the most used data sets in password auditing and penetration testing is the rockyou password set. It contains 14,344,391 passwords procured from a data leak of the company RockYou. Back in 2009, RockYou, a company that made widgets for Facebook and Myspace had its servers hacked, which would not have been nearly as devastating if it wasn't for the fact that RockYou did not practice proper data security measures and stored all of their passwords in plaintext. The hackers immediately released the document to the public. Now, this list is famous enough that it is provided in the default implementation of Kali Linux, in /usr/share/wordlists/rockyou.txt.gz.

Because I am unsure whether the list is sorted by popularity of the password, I also procured a list of the top ten thousand most common passwords from SecList, one of the leading penetration testing companies in the world.

In this list, which is sorted by commonness of the password, If we look at the top 100 passwords, 26 are exclusively digits, 70 are exclusively characters, and only 4 are a combination of digits and numbers. Out of those 70 that are exclusively characters, 60 are English words or some combination of English words, and 10 are easy to remember random sequences of characters, such as the top row of an American keyboard, "qwertyuiop".

The top 10 most common passwords, according to SecList are: password, 123456, 12345678, 1234, qwerty, 12345, dragon, baseball, football.

In addition to this list of passwords, I will use a dictionary of the top 20000 most common english words, for studying dictionary-based attacks. first20hours on Github provides such a list[3].

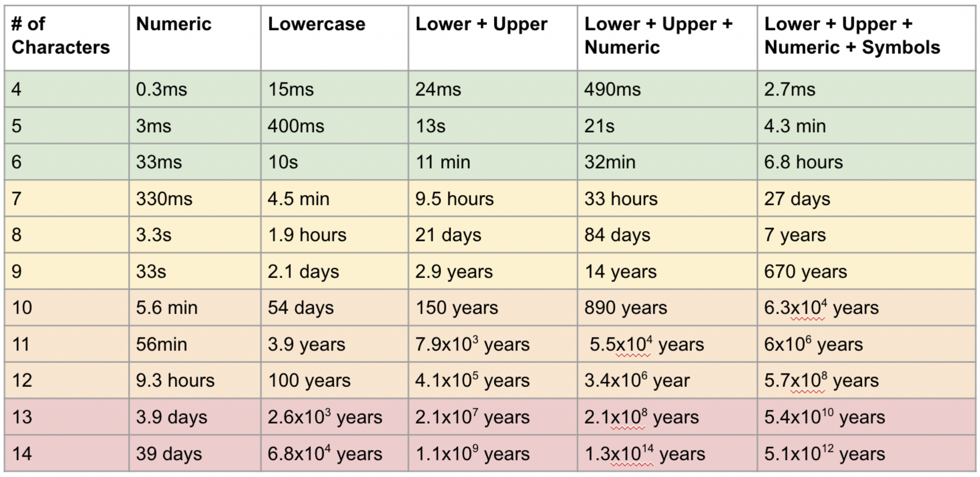
4 Methods

For the coding portion, I implement 4 separate classes in password analysis, and cracking. On the cracking side, I implement a brute force password cracker and a dictionary password cracker. For analysis which is where a great majority of the work was done, I implement a pattern analyzer for the RockYou dataset, and finally a policy-based mask generator, which was the original goal of the project.

4.1 Brute Force Cracker

A brute force cracker works simply by trying every single possible string combination creatable given an alphabet and some constraints(otherwise it will run forever). It is the only cracking method that is 100% guaranteed to guess a password, though it may decades or even eon. The formula for the max time taken to crack a password in this method is:

$$MaxTime = \frac{Possibilities^{PasswordLength}}{AttemptsPerSecond}$$



# of Characters	Numeric	Lowercase	Lower + Upper	Lower + Upper + Numeric	Lower + Upper + Numeric + Symbols
4	0.3ms	15ms	24ms	490ms	2.7ms
5	3ms	400ms	13s	21s	4.3 min
6	33ms	10s	11 min	32min	6.8 hours
7	330ms	4.5 min	9.5 hours	33 hours	27 days
8	3.3s	1.9 hours	21 days	84 days	7 years
9	33s	2.1 days	2.9 years	14 years	670 years
10	5.6 min	54 days	150 years	890 years	$6.3x10^{4}$ years
11	56min	3.9 years	$7.9x10^{3}$ years	$5.5x10^{4}$ years	$6x10^{6}$ years
12	9.3 hours	100 years	$4.1x10^{5}$ years	$3.4x10^{6}$ year	$5.7x10^{8}$ years
13	3.9 days	$2.6x10^{3}$ years	$2.1x10^{7}$ years	$2.1x10^{8}$ years	$5.4x10^{10}$ years
14	39 days	$6.8x10^{4}$ years	$1.1x10^{9}$ years	$1.3x10^{14}$ years	$5.1x10^{12}$ years

Figure 3: An extrapolation of how long it would take to crack passwords of varying lengths and with varying character sets

And so the time scales exponentially with the length of passwords. In Figure 3, we show the results of a brute force cracker, where the values in green are easily crackable using this method, the values yellow, orange, are increasingly dificult to crack until the color red, where as long as the passwords are not just numbers, the password can not be cracked in a lifetime. These values are assuming a GeForce GTX 1080, which can do roughly 30 million attempts every second [10], though there are certainly specialized devices that can perform up to 100 million attempts per seconds.

Because it is so simple to implement in Python, I will show the function that performs basic brute force here:

```
def brute_force(real):
    chars = string.ascii_lowercase + string.digits
    i = 1
    # try all letter combinations up to 10 digits, any higher is just silly
    for password_length in range(11):
        for guess in itertools.product(chars, repeat=password_length):
            i += 1
            guess = ''.join(guess)
            if guess == real:
                print('found in {} guesses.'.format(i))
                return True
```

Brute force crackers can also take advantage of social engineering and the knowledge that many people use personal information in their passwords. For instance, we might refine this basic method to try various combinations that include the victims birthdate or name within the password.

4.2 Dictionary Attack Cracker

A dictionary attack works simply by trying all words in a given dictionary, such as using the RockYou dataset and trying the most popular 10,000 passwords against some account. If you try enough accounts, there will eventually be statistical significance that points towards a very high level of confidence in getting a password from that list, though it will never be 100%. This is simply implemented as:

```
def dictionary_attack(real, dictionary):
    for (i,guess) in enumerate(dictionary):
        if real == guess:
            print('found in {} guesses.'.format(i+1))
            return True
    return False
```

Python's "in" keyword will handle the rest, interating through the list in $O(n)$ time, comparing every option in the dictionary.

Dictionary attacks can be made even more effective with password mangling strategies, which modify chacters in the dictionary list and try those as well. For instance, when trying "password", we could also try "PASSword", or "Pa55w0rd". This is simply done by adding a mangling dictionary like the following:

```
A: 4, /-\, /_\, @, /\, ,
B: 8,|3, 13, |}, |:, |8, 18, 6, |B, |8, lo, |o, j3, ß, ,
C: <, {, [, (, ©, ¢,
D: |), |}, |], |>,
E: 3, £, ₤, €,
F: 7, |=, ph, |#, |", f
G: [, -, [+, 6, C-,
H: #, 4, |-|, [-], {-}, }-{, }{, |=|, [=], {=}, /-/, (-), )-(, :-:, I+I,
I: 1, |, !, 9
J: _|, _/, _7, 9,[1] _), _], _}
K: |<, 1<, l<, |{, l{
L: |_, |, 1, ][
M: 44, |\/|, ^^, /\/\, /X\, []\/][, []V[], ][\\//][, (V),//., .\\, N\,
N: |\|, /\/, /V, ][\\][, , ,
O: 0, (), [], {}, <>, Ø, oh, , ,
P: |o, |O, |>, |*, |°, |D, /o, []D, |7,
Q: O_, 9, (,), 0,kw,
R: |2, 12, .-, |^, l2, , ®
S: 5, $, §,
T: 7, +, 7‘, ’|’ , ‘|‘ , ~|~ , -|-, ’][’,
U: |_|, \_\, /_/, \_/, (_), [_], {_}
V: \/
W: \/\/, (/\), \^/, |/\|, \X/, \\’, ’//, VV, \_|_/, \\//\\//, , 2u, \V/,
X: %, *, ><, }{, )(, ,
Y: ‘/, ¥, \|/, , ,
Z: 2, 5, 7_, >_,(/),
```

While many people may believe that password mangling adds a great degree of security their passwords, I have shown that this is not the case, and a subtle adjustment to a dictionary can easily account for many common mangling strategies. This idea is expounded upon further by Weir et al, who use PCFG's to generate their mangling rules, rather than a simple dictionary[12].

4.3 Group Pattern Password Analysis

I implement a pattern analyzer for the RockYou dataset that provides a series of stats on the passwords in the list, including length, most common masks, character makeup, and a differentiation between simple and complex masks. This method works by running individual analysis on each password in a list, keeping a running list of passwords and masks seen and noting the areas of similarities between passwords that have already been visited and the current password, not unlike the algorithm for Breadth First Search or Depth First Search, though implemented iteratively. The outputted mask patterns are then used in the policy mask generation in order to allow for probalistic ordering, such that the most common masks are generated at the top, and the least common masks are generated last. When discussing this more in the Results section, we will break down what each piece of analysis means, and how it provides useful information in the the patterns of how people's passwords.

4.4 Policy Mask Generation

A mask, as defined in the standard provided by HashCat, an extremely popular penetration testing tool available with Kali Linex, follows the following rules:

- ?l = lowercase letters: abcdefghijklmnopqrstuvwxyz
- ?u = uppercase letters: ABCDEFGHIJKLMNOPQRSTUVWXYZ
- ?d = digits: 0123456789
- ?s = symbols: `` <<space>>"#$%&'()*+,-./:;<=>?@[\]^_`{|}~! ``

Using these masks, we can generate all possible combinations of these 4 rules up to some length of string. For instance, the string "pass12" is masked as "?l?l?l?l?d?d", which is the same mask as "pass23", and any other word in the language defined by 4 lowercase letters followed by 2 digits. Masks form a language similar to how a regular expression can form a language, such as "a.*1" forming the language of all strings that start with a, have any number of characters in the middle, and end with a 1.

The importance of noting that there are 26 lowercase letters, 26 uppercase letters, 10 digits, and 33 symbols is it's use in estimating password cracking times. For instance, we know a 5 character password that is all digits has $10^5 = 100,000$ possibilities, while a 5 character password that is all symbols could have $33^5 = 39135393$ possibilities, which would take 391 times longer to crack.

NOTE: There are many websites that do not allow all of these symbols, as without proper sanitation practices, passwords that contain escape characters for javascript and html, such as quotation marks, "¡", and "¿", and backslashes allow for cross-site scripting. However, there are many websites that do allow these symbols, either because they do not have have any cross-site scripting protections, or because they are well-santizing the password field, so we consider all 33 symbols in our analysis.

This is an especially important consideration. Most websites that require passwords, such as to access social media or online banking, have stringent requirements to include at least 1 lowercase letter, at least one uppercase letter, at least 1 digit, and at least 1 special character. As such, there is no point in trying passwords that do not have these qualifications.

Using a series of passed in rules, we output a list of masks that we would like to try. These masks can then be used in a Mask Attack, such as the one provided by HashCat[4].

5 Results & Discussion

While we have four methods, each which could have results, the brute force cracking results and dictionary based results are used in the context of proving the correctness of my pattern analysis, and policy-based mask generator, so those are the results we will analyze, as the patterns parsed from the output is what is ultimately used to refine the password cracking methods.

5.1 Pattern Analysis

On the top 101 passwords, we get the following:

```
LENGTH:
6: 48% (49)
7: 18% (19)
8: 17% (18)
9: 7% (8)
5: 3% (4)
10: 2% (3)

CHARACTER SETS:
lowercase: 85% (86)
numeric: 11% (12)
lowercase-numeric: 1% (2)
uppercase: 0% (1)

PASSWORD COMPLEXITY:
Digits: min:0 max:10
Lowercase: min:0 max:10
Uppercase: min:0 max:8
Special: min:0 max:0

SIMPLE MASKS:
string: 86% (87)
digit: 11% (12)
stringdigit: 1% (2)
```

```
COMPLEX MASKS:
?l?l?l?l?l?l: 41% (42)
?l?l?l?l?l?l?l: 17% (18)
?l?l?l?l?l?l?l?l: 15% (16)
?d?d?d?d?d?d: 5% (6)
?l?l?l?l?l?l?l?l?l: 4% (5)
?l?l?l?l?l: 2% (3)
?d?d?d?d?d?d?d?d: 1% (2)
?l?l?l?l?l?l?l?l?l?l: 1% (2)
?d?d?d?d?d: 0% (1)
?u?u?u?u?u?u?u?u: 0% (1)
?d?d?d?d?d?d?d: 0% (1)
?d?d?d?d?d?d?d?d: 0% (1)
?l?l?l?d?d?d: 0% (1)
?l?l?l?l?l?l?l?l?d: 0% (1)
?d?d?d?d?d?d?d?d?d?d: 0% (1)
```

NOTE: Some of the percentages may look off by 1% due to the fact that there are 101 passwords analyzed and the percentage is rounded down via truncation of a float to int.

NOTE: The full pattern analysis, containing all 14 million passwords and 145,000 complex masks can be seen in the provided "patterns.txt" file included in Guha-Final.tgz. For the sake of understanding the general patterns without getting bogged down in hundreds of thousands of lines of unique masks, we will perform our analysis on the 101 passwords shown above.

Let's dive into the useful and a discussion of each of these five pieces of information.

5.1.1 Length

We can see that 83% of passwords are between a length of 6 and 8. Because brute crackers can easily guess all combinations of passwords of length 5 and less, we can reduce our work load considerably in guessing passwords simply by targeting our mask generator at the 6-8 length range.

5.1.2 Complexity

Here, we quickly analyze the most and least complex password in our dataset. For instance in our run of 100 passwords, the longest password was 10 digits, the password with the most uppercase characters had 8, the password with the most lowercase characters had 10, the password with the most digits had 10, and there were no passwords with special characters. Because one of the passwords in the dataset is the empty password, the minimum is always 0 for all requirements, though this certainly may not be the case on a dataset with exclusively complex passwords.

5.1.3 Character Breakdown & Simple Masks

85% of passwords are some number of lowercase letters exclusively. This further allows us to hone our brute force attacker as well as our mask generator, since we can reasonably guess 85% just looking at combinations of 26 digits.

The simple mask is a more complex interpretation of the charter breakdown, which simply shows us what set of characters a percentage of passwords belong to. Simple masks show us the consecutive character sets. For example, the string "1a1a1a" would have a simple mask of "stringdigitstringdigit-stringdigit", and belong to the charset of "loweralphanum", since it has only characters in loweralpha, and characters in num.

You may also notice that in the top 100 passwords, there are no uses of uppercase or symbols, and it happens in less than 1% of passwords.

5.1.4 Complex Masks

These are proper mask representations of the passwords, along with the percentage that follow the pattern. In the top 100 passwords, there were 15 unique masks, but 7 masks only had one password that matched them, and 5 masks only had less than 6 passwords that matched them. That means just 3 masks contained 73% of passwords.

5.2 Policy Mask Generation

Using the information provided by our pattern analysis, we can tune the precise settings that will get be generated by our policy mask generator. From our analysis and statistics work, we know we can capture a majority of passwords with the following assumptions:

1. passwords are most likely to be between 6-8 characters in length
2. prioritize digit only passwords, and lowercase only passwords
3. if a password will have a capital letter, it will be at the beginning of the password
4. if a password will have a symbol, it will be at the end of the password.
5. passwords with letters and digits will have the digits at the end of the password.

This tells us that when choosing a new password that meets a given policy, the human psychology is not to create a new, unique password that meets the criterion, it is instead to modify an existing password they use to meet the criterion as quickly as possible. That is, we note that very little thought goes into password choosing, and this can be exploited. One way to exploit such knowledge is to generate a probabilistic ordering of masks, such that the most expected masks come first, which might looks something like this :

```
?l?l?l?l?l?l
?l?l?l?l?l?l?l
?l?l?l?l?l?l?l?l
?d?d?d?d?d?d
?d?d?d?d?d?d?d
?d?d?d?d?d?d?d?d
?l?l?l?l?d?d
?l?l?l?l?l?d
...
```

However, where the policy mask generator shines is when considering password requirements set by companies such as social media websites and online banks. In the most secure cases, we must specify at least 1 lowercase letter, at least one uppercase letter, at least 1 digit, and at least 1 special character, and have a length of at least 8. Running this through the mask generator, we get a list of masks including the following:

```
?u?l?l?l?l?l?d?s
?u?l?l?l?l?d?d?s
?u?l?l?l?d?d?d?s
?d?d?d?d?d?u?s?l
?d?d?d?d?d?s?l?u
?d?d?d?d?d?s?u?l
?d?d?d?d?l?d?u?s
...
```

Most usefully, however, my mask generator outputs an estimate of how long it will take to run each of the masks:

```
Total Masks:  65536 Time: 76 days, 18:50:04
Policy Masks: 40824 Time: 35 days, 0:33:09
```

We can see that if we ran all $4^8 = 65536$ possibilities(given by a string of length 8 having 4 possibile string masks for each character), it would take 76 days, but by using my policy-based masked generator, we can reduce the number of masks attempted by 38% to 40824, and more impressively, reduce the ammount of time it takes to make these guesses by 54%, to 35 days. This time could be reduced even further if we account for Wang et al. and her work in Zipf's laws connection to passwords, where if we estimate the liklihood of a password being within the first 100 or 1000 masks, we could say that on average, it would take less than even 35 days, which is based on trying every possible policy mask.

6 Future Work

There are three, distinct directions that I believe this research could be a first step towards, each interesting and exploring a unique topic in password analysis.

Password Generation To further expand on my work in password security using parsing techniques, I believe it would be useful to take this analysis and use it to create better passwords. That is, write a better password generator that is specifically designed to avoid the common patterns that can be accounted for in my software design, such as leet, mangling, salting, dictionary approaches, or most relevantly, masking. At the same time, it is important to consider the utility of a password, as an extremely secure password such as "26DczHpaRKj@vE" has almost no utility and would never get used. Instead, a balance must be struck between the usefulness of a password and it's security. This analysis here helps us further that research, but until a method is devised that provides perfectly secure passwords.

Confidence Intervals % Hypothesis Testing Using the patterns deciphered and obtained in this paper, there should be a solvable statistical problem in deducing the liklihood of password guessing schemes, such as being able to say with some percent of confidence that between $\mu - \sigma$ and $\mu + \sigma$ passwords will be cracked using a given set of masks.

The language barrier The password list from RockYou analyzed here contains millions of substrings or direct English words, and it would stand to reason that in another country such as Germany, France, Spain, etc. that many passwords would contain words from their respective native languages. Running the group password pattern analyzer on these culture-specific datasets could reveal either that password choosing behaviour varies from country to country and language to language, or that there are some ubiquitous, intrinsically human patterns in password choosing behaviour that can be exploited.

7 Conclusions

I was able to create 4 distinct pieces of software that are useful in password analysis, password cracking, and mask generation. I analyzed a list of over 14 million passwords, and was able to do basic rule induction as to how people create their passwords. I showed how these improvements can have drastic results on the time it takes to crack a given password in one case improving results by 54%. Knowing these common psychological patterns people choose allows companies to better tailor their password requirements to prevent mask attacks that can easily account for this psychology. Lastly, I showed that while this work can easily stand on it's own in offering useful insights and providing reader's with the most common flaws and how to prevent their passwords being easily hacked by malicious actors, it also serves as an important first step in other work in password security that certainly use the additional research.

References

[1] Sudhir Aggarwal, Shiva Houshmand, and Matt Weir. *New Technologies in Password Cracking Techniques*, pages 179–198. Springer International Publishing, Cham, 2018.

[2] John D Cook. Actual password distribution follows power law, Sep 2018.

[3] first20hours. first20hours/google-10000-english, Jul 2016.

[4] HashCat. hashcat advanced password recovery.

[5] Shiva Houshmand and Sudhir Aggarwal. Using personal information in targeted grammar-based probabilistic password attacks. In Gilbert Peterson and Sujeet Shenoi, editors, *Advances in Digital Forensics XIII*, pages 285–303, Cham, 2017. Springer International Publishing.

[6] W. Li and J. Zeng. Leet usage and its effect on password security. *IEEE Transactions on Information Forensics and Security*, 16:2130–2143, 2021.

[7] Y. Li, H. Wang, and K. Sun. A study of personal information in human-chosen passwords and its security implications. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, 2016.

[8] Simon Marechal. Advances in password cracking. *Journal in Computer Virology*, 4:73–81, 02 2008.

[9] B. Medlin, Douglas May, and Ken Corley. Healthcare employees and passwords: An entry point for social engineering attacks. 2011.

[10] University of South Wales. Brute force password hacking: How long will it take to brute force a password, Jul 2020.

[11] D. Wang, H. Cheng, P. Wang, X. Huang, and G. Jian. Zipf's law in passwords. *IEEE Transactions on Information Forensics and Security*, 12(11):2776–2791, 2017.

[12] M. Weir, S. Aggarwal, B. d. Medeiros, and B. Glodek. Password cracking using probabilistic context-free grammars. In *2009 30th IEEE Symposium on Security and Privacy*, pages 391–405, 2009.