Password Cracking Techniques and Probability

Louise Xu, Elle Wen

December 2022

1 Introduction

When a police officer arrests a bad guy but the only evidence that can convict the guy of actual crimes is stored in a file protected by a password. Obviously the bad guy would refuse to give out the password. To access the file, a smart police officer already bypassed the system that protecting the file and gained direct access access to the encrypted files – so now the police officer has unlimited chances to guess and test the actually password and works under what we called a "offline attack". They now needs an effective password cracker to crack the password.

A general password cracker works as follows: it generates multiple password guesses and tries each one to see if it matches the target password, until it finds a match. Many passwords have been leaked in the past few years due to data breaches. For example, the RockYou password list consists of over 32.6 million passwords from the website www.rockyou.com, that were released to the public after a successful SQL injection attack against the password database in December 2009 [1]. These password database can be used to help us construct more efficient password crackers by training on those data and learning common password patterns. Although the password cracker would be trained to incorporate site-specific aspects, this adaptability is actually useful since password distributions are different for different sites. For example, the password `rockyou` is only popular in the RockYou service. As such, using information from a specific training set would enable the password cracker to perform better in the corresponding website service. Moreover, studying effective password cracking method would also help users eliminate weak passwords and choose a strong password.

This report will discuss two primary categories for password cracking: brute force attacks and dictionary attacks in detail. We will introduce a few standard attacking methods as well as a probabilistic method under each category.

2 Brute Force Attacks

2.1 Pure Brute Force

The simplest brute force attack is to try every single possible password given a character set. Let a character set Σ denote the set of characters that are being used to generate passwords. A key space is the set of all possible passwords. Suppose $|\Sigma| = x$ is the size of the character set and n is the maximum length of the password, then the pure brute force method would search exhaustively through the key space of size $\sum_{k=0}^{n} x^k$. Some example character sets are shown below.

The order in which the pure brute force attack makes guesses depends on the inherent order of the given character set. For example, if the loweralpha character set is given which has an implicit alphabetic order, then an attack guessing passwords of length six would start by guessing

Name	Character Set
numeric	0123456789
loweralpha	abcdefghijklmnopqrstuvwxyz
loweralpha-numeric	abcdefghijklmnopqrstuvwxyz0123456789
mixalpha	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
mixalpha-numeric	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
mixalpha-numeric-all-space	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!@#$%^&*()-_+=~`[]{}|\:;"'<>,.?/

Figure 1: Example Character Sets for Pure Brute Force Attacks

"aaaaaa" and end with "zzzzzz". There are two ways of making this sequence of guesses. For password crackers that makes guesses in the *right incrementing order*, the sequence would be "aaaaaa, aaaaab, aaaaac ...", while those that make guesses in the *left incrementing order* guess in the sequence "aaaaaa, baaaaa, caaaaaa".

The pure brute force method searches through the entire key space, so it is guaranteed to crack the password. However, it is not the most efficient method, since it only follows the default order of the character set. It doesn't take advantage of any additional knowledge that we may have about common password traits.

2.2 Letter Frequency Analysis

One improvement that can be made on pure brute force is to guess using the frequency of characters appearing in a training set instead of the default order. For example, a popular training set is the the novel Moby Dick, and the resulting letter distribution is commonly cited as the standard letter distribution of the English language. Although this generalization might be slightly problematic, it shows how we can extract information from training data to aid password cracking. In addition to the overall letter distribution, we could also generate a distribution for the first and last letters of the password. These distributions would usually be different from the overall distribution, since they are able to capture common human behaviors such as capitalizing the first letter and adding digits to the end. Therefore, this additional information on letter distributions is helpful for the password cracking process since it allows the password cracker to learn certain patterns from real-life passwords and thus able to generate passwords that are more likely to be chosen by humans.

As a side note, letter frequency analysis is more commonly used in cryptoanalysis, since it can search for frequent letters in the encrypted text and match with frequent letters in the English language. Nonetheless, letter frequency analysis does increase the effectiveness of pure brute force, as it attempts to crack a majority of the target passwords while only searching a small fraction of the total key space.

2.3 Whole-string Markov method

The Markov method is another step up against the letter frequency analysis. The presumption is that humans don't choose their passwords randomly, and that adjacent letters in human-generated passwords are not chosen independently. For example, the letter "e" is much more likely than any other letter to follow the letters "th". We can use a Markov model to model this idea that

the probability of the next character depends on the previous character(s). Once we have trained the markov model on a training set, we can compute the probability of any candidate password guess. Then, we would generate password guesses starting from the most probable passwords, and continuing in decreasing probability order.

2.3.1 Markov model: 391 Calculations

A Markov model is a stochastic model which describes a sequence of possible events (or states) in which the probability of each event depends on a subset of previous events. More specifically, an n-th order Markov model only considers n previous states. We will focus on first order Markov models where the probability of a future state depends only on the current state, since higher order Markov models can be simplified to the first order Markov model. To better illustrate this reduction, we start by explaining a simple first order Markov model with two states A and B shown in Figure 2. The transition edges between state encode the conditional probability of observing a certain future state given the current state.

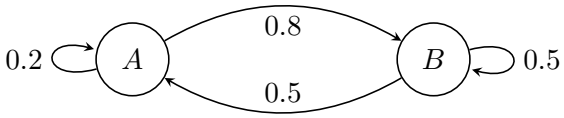

Figure 2: First order Markov model with states A, B

More formally, the conditional probabilities are

$$P(A|A) = 0.2, \quad P(B|A) = 0.8, \quad P(A|B) = 0.5, \quad P(B|B) = 0.5.$$

Now suppose we have a third order Markov model, where the probability of observing a future state depends on three previous states. The conditional probability $P(A|BCD)$ can be encoded as a transitional edge from state BCD to state CDA. Hence, we can still encode this higher-order model as a first order Markov model, except the new states would be ordered 3-tuples of the original states.

Continuing with the simple Markov model, the overall probability of seeing a sequence of ABA can be computed by the product of the conditional probabilities:

$$P(ABA) = P(A)P(B|A)P(A|B) = P(A)(0.8)(0.5) = 0.4P(A).$$

Here we multiply the probabilities because of the independence assumption in Markov models. The first order Markov model assumes that the transition probability to a future state is only depend on the current state, hence independent of any states observed in the past. Therefore, we can compute the conditional probability of observing a sequence like $P(BA|A)$ by multiplying the respective conditional probabilities. The initial probability of observing an A may be derived from a separate initial distribution. Alternatively, it may be easily resolved by adding a starting state S with corresponding probability values. Then the corresponding Markov model would have three states: A, B, and S (Figure 3). For clarity, edges with probability zero are not drawn out.

Then the probability of observing the sequence ABA would be

$$P(ABA) = 0.4(0.6) = 0.24.$$

Instead of viewing the probabilities as conditional probabilities of the future states, we can also interpret them as the probability of observing a current state and future state pair. In other words,

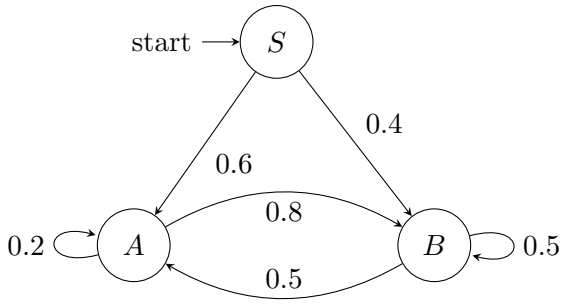

Figure 3: First order Markov model with states A, B, S.

we would have probabilities

$$P(AA) = 0.2, \quad P(AB) = 0.8, \quad P(BA) = 0.5, \quad P(BB) = 0.5.$$

We would use this notation for discussing password cracking models, since they are easier to keep track of in the training set. We call this the probability of digrams, which are ordered pairs of characters. Hence an n-th order Markov model is also called an $(n+1)$-gram Markov model.

2.3.2 N-gram database

In the context of password cracking, we assume a password is made up of a sequence of characters $(c_1, c_2, \ldots, c_l)$. In a n-gram Markov model, each possible n-gram is assigned a probability. These probabilities are obtained from an n-gram database which records the frequency of each n-gram in the the training set. An algorithm for constructing the n-gram database is given below.

1. Initialize all counts to 0.
2. Whenever a password $c = (c_1, c_2, \ldots, c_l)$ is read, it is decomposed into $l-n+1$ many n-grams. The database is then updated by incrementing the counts of those n-grams by 1.

This process is executed once only so we aren't worried too much about it's efficiency. Once the n-gram database is constructed, we can compute the conditional probabilities using the frequency counts. For example, the probability of the string "password" is computed with the following formula using a 5-gram Markov model.

$$P(password) = P(p)P(a|p)P(s|pa)...P(d|swor).$$

As an example, the conditional probability $P(d|swor)$ is computed by $\frac{count(sword)}{count(swor)}$ where $count(swor) := \sum_{x \in \Sigma} count(sworx)$. Therefore, we have constructed a **probability mass function** that assigns any candidate password string a probability. In other words, we can use the trained Markov model to compute the probability of any given password guess. Now the only problem that remains is how do we output password guesses in decreasing likelihood order, so that we can guess the most likely passwords first. A naive approach would be to compute probabilities for all possible passwords and then sort them in decreasing probability order. Obviously this would be too expensive in terms of both time and storage, so we seek a more efficient method to generate password guesses in decreasing probability order.

2.4 Password strength estimator

Before going into the password generation process, let's look at a problem closely related to password guessing: estimating the strength of a password. This is of central importance for the operator of a site to ensure a certain level of security. We can use password crackers to test the strength of a password and then reject weak passwords; such mechanisms are often called **proactive password checkers**. One way of estimating the strength of a password is by taking the negative logarithm, so passwords with higher probability will be assigned lower strengths.

$$f(c) = -\log_2(\prod_{i=0}^{n} P(c_i|c_{i-n+1}, \ldots, c_{i-1})).$$

This trick of taking the logarithm to convert password probabilities into a notion for password strength is central to the following password generation algorithm for Markov models.

2.4.1 Password Generation

The Markov model-based password cracker first proposed by Narayanan and Shmatikov in 2005 suggests an indexing method which is implemented by famous password cracker John the Ripper in its Markov mode. However, this method does not generate passwords in order of decreasing probability. In 2015, Duermuth proposed the improved Ordered Markov ENumerator (OMEN) algorithm, which enumerates passwords with (approximately) decreasing probabilities.

On a high level, the OMEN algorithm discretizes all password probabilities into a number of bins, and iterates over all those bins in order of decreasing likelihood. For each bin, it finds all passwords that match the probability associated with this bin and outputs them. This discretization is achieved through the use of logarithms. For simplicity, we will illustrate the algorithm with a 3-gram Markov model.

Suppose we are interested in finding all strings c that are above a probability threshold θ,

$$P(c) = P(c_1, c_2) \prod_{i=3}^{n} P(c_i|c_{i-2}, c_{i-1}) \geq \theta.$$

By taking the logarithm, we turn the product of the decomposed probabilities into a sum,

$$\log P(c) = \log P(c_1, c_2) + \sum_{i=3}^{n} P(c_i|c_{i-2}, c_{i-1}) \geq \log \theta.$$

We can round the log probability $\log \theta := \mu$ to the nearest multiple of μ_0, where μ_0 is chosen appropriately so that we have a sufficient number of different values. We call these rounded values the level of the password. The level denotes the strength of the password, where a higher level represents a stronger password. In other words, the password probabilities are discretized into several bins, where each bin corresponds to a certain level. Since the probabilities are less than 1, the logarithm would return negative values. Hence the levels can take values $0, -1, \ldots, -(nbLevel - 1)$ where nbLevel is the total number of levels. This is a hyperparamter that can be adjusted to tune the level of discreteness that we want. We would want to choose an appropriate rounding mechanism where the highest probability is rounded to the level 0, and the total number of levels is $nbLevel$. A higher $nbLevel$ means better accuracy since the passwords are placed into smaller bins, but it also means a longer run time.

As we can see in the above formula, the log probability of a password is the sum of decomposed log probabilities. Thus in practice, we assign levels to the individual n-grams and compute the

overall level of a password by taking the sum. More specifically, we take the logarithm of all n-gram conditional probabilities and discretize them into levels (denoted μ) according to the formula

$$L(x) = \text{round}(\log(c_1 P(x) + c2))$$

where c_1 and c_2 are chosen such that the most frequent n-grams get a level of 0. The offset c_2 is included so that n-grams that did not appear in the training are still assigned a small probability. The parameters c_1, c_2 are adjusted to get the desired number of levels ($nbLevel$).

The Enumeration algorithm The enumPwd algorithm enumerates all passwords of a given length ℓ and level μ, and the process for a 3-gram Markov model proceeds as follows.

1. First compute all vectors $a = (a_2, \ldots, a_\ell)$ of length $\ell - 1$ where each entry is the level of its decomposed probabilities. Each entry a_i is an integer in the range [0,nbLevel-1], and the sum of all entries is μ. The vector is of length $\ell - 1$ because we are using 3-grams, so we need $\ell - 2$ transition probabilities and 1 initial probability to determine the probability for a string of length ℓ. To illustrate this with an example, the probability of "password" with length $\ell = 8$ is computed as follows:

$$P(password) = P(pa)P(s|pa)P(s|as)P(w|ss)P(o|sw)P(r|wo)P(d|or).$$

2. For each such vector a, select all 2-grams x_1x_2 whose probabilities match level a_2. For each of these 2-grams, it iterates over all x_3 such that the 3-gram $x_1x_2x_3$ has level a_3. Next, for each of these 3-grams, it iterates over all x_4 such that the 3-gram $x_2x_3x_4$ has level a_4, and so on, until the desired length is reached. It outputs the final password candidate that fulfills all of the requirements. It doesn't output anything if no password fulfills the requirements of vector a.

In the end, this process outputs a set of candidate passwords of length ℓ and level μ. A more formal description is also presented in Algorithm 1.

Algorithm 1 Enumerating passwords for level μ and length ℓ (here for $\ell = 4$)

function ENUMPWD((μ, ℓ))
  for each vector $(a_2, \ldots, a_\ell)$ with $\sum_{i=2}^{\ell} a_i = \mu$
  and for each $x_1x_2 \in \Sigma^2$ with $L(x_1x_2) = a_2$
  and for each $x_3 \in \Sigma$ with $L(x_3|x_1x_2) = a_3$
  and for each $x_4 \in \Sigma$ with $L(x_4|x_2x_3) = a_4$
  output $x_1x_2x_3x_4$.

Password Generation Example We will demonstrate the algorithm with a brief example. For simplicity, we consider passwords of length $\ell = 3$ over a small alphabet $\Sigma = \{a, b\}$, where the initial probabilities have levels

$$L(aa) = 0 \quad L(ba) = 1,$$
$$L(ab) = 1 \quad L(bb) = 0,$$

and the transitions have levels

$$\begin{aligned} L(a|aa) &= 1 \quad L(b|aa) = 1 \\ L(a|ab) &= 0 \quad L(b|ab) = 2 \\ L(a|ba) &= 1 \quad L(b|ba) = 1 \\ L(a|bb) &= 0 \quad L(b|bb) = 2 \end{aligned}$$

1. Starting with enumPwd($\mu = 0, 3$), only the vector (0, 0) is given. This yields the password `bba` only (the prefix "aa" matches the level 0, but there is no matching transition with level 0).

2. Next enumPwd($\mu = -1, 3$) gives the vector (-1, 0) as well as the vector (0, -1). The former yields `aba` (the prefix "ba" has no matching transition for level 0) while the later yields `aaa` and `aab`.

3. Next enumPwd($\mu = -2, 3$) gives three vectors: (-2, 0) yields no output (because no initial probability matches the level -2), (-1, -1) yields `baa` and `bab`, and (0, -2) yields `bba`.

4. And so on for all remaining levels.

Notice that the *enumPwd* algorithm itself does not guarantee anything on the order of the password outputs for a given length and level. It merely enumerates all passwords that satisfy the bin condition, but the true probability of the passwords within the same bin may vary slightly. Nonetheless, we consider this variability to be acceptable, since all of the passwords enumerated by one execution of enumPwd are considered to be approximately the same strength, so we are not too concerned about the specific order in which they are enumerated. The more important thing is that, when called with a given length and decreasing levels, we are able to enumerate passwords with approximately decreasing probability. That being said, a higher number of levels will allow for more finely discretized probabilities to model the order of true probability closer since the bins are narrower. This can potentially increase accuracy though at the expense of higher run times.

The OMEN algorithm The OMEN algorithm calls the enumeration algorithm, enumPwd(ℓ, μ), iteratively with appropriate choice of the two parameters. With a selected length parameter, it is clear that we should start enumerating passwords from the highest level. However, the length parameter is challenging to select, since the length frequency in the training set is not a good indicator of how often a specific length should be guessed. For example, we may find a training set with the same number of passwords of length 7 and of length 8. However, passwords of length 7 should be guessed first because the search space is smaller. Here we introduce a new notion of **success probability**, which is the ratio of successfully guessed passwords over the total number of generated password guesses. While Dürmuth's et al's paper doesn't specify this, we assume that the successfully guessed passwords refer to guessed passwords that were also present in the training set. Therefore, we use a priority queue structure to implement an adaptive algorithm that keeps track of the success ratio of each length and schedules to guess more passwords of those lengths that were more effective. For a max priority queue, new elements are always inserted to the end of the queue. When elements are popped, or removed, the elements with the maximum priority are always the first to be removed. More precisely, the OMEN algorithm works as follows:

1. For all m length values of ℓ (if we consider lengths from 3 to 20, then $m = 17$), execute enumPwd(0,ℓ) and compute the success probability $sp_{l,0}$. This probability is computed as

the ratio of successfully guessed passwords over the total number of generated password guesses of length ℓ.

2. Build a max priority queue M with the m triples (sp,level,length) ordered by the success probabilities. (Since this is a max priority queue, the element with the largest success probability is always removed first.)

3. Remove the element with the largest success probability $(sp_0, level_0, length_0)$. Run enumPwd $(level_0 - 1, length_0)$, compute the new success probability sp^*, and add the new element $(sp^*, level_0 - 1, length_0)$ to M.

4. Repeat steps 3 and 4 until M is empty or enough guesses have been made.

First of all, we can notice whenever one element is popped from the queue, another element with the same length and decremented level is added. Since the levels represent the strength, or the probability, of passwords, this means that the inserted elements will always produce passwords with smaller probability than the removed element. Since a priority queue always outputs in maximum order, it guarantees that the specific instantiations of the enumerating algorithm for a given length are always called in decreasing probability.

Unfortunately, this does not necessarily mean that the OMEN algorithm generates passwords of all lengths in order of decreasing probability. We must also consider instantiations of enumPwd with different lengths. Even though the OMEN algorithm tries to optimize a password generating strategy over different password lengths by tracking success probabilities for each length and prioritizing lengths with high success probability, we must keep in mind that high success probability is not equivalent to high password probability. Consider a situation where enumPwd(ℓ_i, μ_i) has higher success probability than enumPwd(ℓ_j, μ_j) hence it gets popped and executed first. However, it is sill possible to for enumPwd(ℓ_j, μ_j) to enumerate a password with a higher probability than some password enumerated by enumPwd(ℓ_i, μ_i). If we were able to assert that high success probability necessary implies generating passwords with higher probability, than this would be a situation similar to the PCFG model addressed in section 3. That method also uses a priority queue, but it is able to output password guesses with true decreasing probability order. Nonetheless, OMEN outputs is still capable of guessing in order of approximately decreasing frequency and dramatically improves real-world guessing speed.

3 Dictionary Based Attacks

3.1 Overview

The basic idea of dictionary based attacks is simple: the attacker first creates a dictionary comprised of all possible words they suspect the target may have used in their password, and then applies mangling rules that modifies those words for a better match with the target's password. Mangling rules are created based on human's habits of creating passwords such as capitalizing the first letter or putting digits after characters. An example of the first ten mangling rules used in one of the most popular software for password cracking John the Ripper is shown in Figure 4.

In reality, the attacker often starts by using a very small input dictionary of highly probable words, and apply many complicated word mangling rules to them. If the attack is not successful at this stage, the attacker then use a much larger input dictionary and apply the same mangling rules as before. However, there are two problems. First, higher probability dictionary word with low probability mangling rule is always guessed first. But an attacker may want to try a low

Rule #	Mangling Rule
1	Try words as they are
2	Lowercase every pure alphanumeric word
3	Capitalize every pure alphanumeric word
4	Lowercase and pluralize pure alphabetic words
5	Lowercase pure alphabetic words and append '1'
6	Capitalize pure alphabetic words and append '1'
7	Duplicate reasonably short pure alphabetic words (fred -> fredfred)
8	Lowercase and reverse pure alphabetic words
9	Prefix pure alphabetic words with '1'
10	Uppercase pure alphanumeric words

Figure 4: Example Mangling Rules from John the Ripper

probability dictionary word with a high probability mangling rules (eg. snake1) before they try a high probability dictionary word with low probability mangling rules (eg. p@aSwoRd392). Second, it is hard to accurately capture all of the knowledge for how people normally create passwords and also hard to optimize running time and memory using a rule based dictionary attack. Take Rule 8 in the table for example. People rarely reverse words when creating passwords, but the resulting words from that rule are tested before other words of higher probability. Thus, instead of discussing dictionary attack based on mangling rules that mimic the ways people create passwords, we focus on an adapted dictionary based attack technique that formally employees the concepts in probability in the rest of this section.

3.2 Using Probabilistic Context-Free Grammars

3.2.1 Preprocessing

In the preprocessing phase, probability information needed for building a probabilistic context-free grammars (PCFG) is gathered from the training set. We need to define some terminologies first.

Let an *alpha string* be a string that contains only alphabet symbols, a *digit string* be a string that contains only digits, and a *special string* be a string that contains only non-alphabet and non-digit symbols. In context-free grammars, we denote alpha string as **L**, digit string as **D**, and special string as **D**. Also we indicate the length of the string with its subscript. For example the passwords "$password123" can be expressed as $\mathbf{S_1L_8D_3}$, which is defined as a base structure. A pre-terminal structure refers to the string obtained after filling in specific values for the **D** and **S** parts of the base structure (eg."$\$L_8123$"). A terminal structure is defined as the final guess generating by the probabilistic context-free grammars after filling in the **L** parts of the pre-terminal structure (e.g. "$password123").

The first preprocessing step is to derive all the base structures and their associated probabilities of occurrence in the training set. And the next step is to derive the probability of digit strings and special strings appearing in the training set. For example, if we define a **random variable** D_1, a possible **probability mass function** is showed in Figure 5, with the values in the first column and the corresponding percentage probability in the third column.

1 Digit	Number of Occurrences	Percentage of Total
1	12788	50.7803
2	2789	11.0749
3	2094	8.32308
4	1708	6.78235
7	1245	4.94381
5	1039	4.1258
0	1009	4.00667
6	899	3.56987
8	898	3.5659
9	712	2.8273

Figure 5: Example pmf of random variable D_1

Note that we decide to not calculate the probability for alpha string. This is because the alpha string consists of the most important part of a password, so the set of alpha strings that users may employ in their passwords is much larger than those present in the training set.

3.3 Constructing Probabilistic Context-Free Grammars

A context-free grammar is defined as $G = (\mathbf{V}, \mathbf{\Sigma}, \mathbf{S}, \mathbf{P})$ where $\mathbf{V}$ is a finite set of variables (such as S_1, D_2, L_3 etc.), Σ is a finite set of terminals (such as $1, 5, \$$ etc.), $\mathbf{S}$ is the start variable, and $\mathbf{P}$ is a finite set of production rules of the form $\alpha \to \beta$ where α is a single variable and β is a string consisting of variables or terminals. The language of G is collection of the strings that can be derived from the starting variable following the rules.

A probabilistic context-free grammar simply have probabilities associated with each production so that for a left-hand side variable, the probability for all its associated right-hand side productions add up to 1. In our grammars, we only use variables L_n, D_n, S_n which are respectively called alpha variables, digit variables, and special variables. The start variable is always S. The first production rule is S producing all the base structures with its associated probability derived in the preprocessing stage. A string derived from the start symbol is called sentential form. The probability of a sentential form is just the product of the probabilities of all the production rules used in its derivation. In our preprocessing stage, we actually already obtain a probabilistic context-free grammar. See an example in Figure 6. Then the probability of a sentential form "$L_3$4!" is

$$S \to L_3 D_1 S_1 \to L_3 4 S_1 \to L_3 4!$$

$$P(L_3 4!) = P(S \to L_3 D_1 S_1) P(D_1 \to 4) P(S_1 \to !) = 0.25(0.60)(0.65) = 0.0975.$$

Note that because we did not calculate the probability for alpha string, our grammars can only get to pre-terminal structures with alpha variables(e.g. $L_3$4!). Given a pre-terminal structure, we use a dictionary to derive the terminal structure. For example, if we have a dictionary $\{cat, hat, apple\}$ and the pre-terminal structure is $L_3$4!, the two guesses (or terminal structures)

LHS	RHS	Probability
$S \rightarrow$	$D_1L_3\,S_2D_1$	0.75
$S \rightarrow$	$L_3D_1S_1$	0.25
$D_1 \rightarrow$	4	0.60
$D_1 \rightarrow$	5	0.20
$D_1 \rightarrow$	6	0.20
$S_1 \rightarrow$	!	0.65
$S_1 \rightarrow$	%	0.30
$S_1 \rightarrow$	#	0.05
$S_2 \rightarrow$	$$	0.70
$S_2 \rightarrow$	**	0.30

Figure 6: Example a probabilistic context-free grammar

are $\{cat4!, hat4!\}$. The order that we test the guesses is we first choose the highest probability pre-terminal structure and fill in all relevant dictionary words, and then choose the next highest pre-terminal structure and fill in all relevant dictionary words, etc. An alternative approach is to assign probabilities to alpha strings in various ways and fill in words according to order of decreasing probability, but we will not discuss these approaches since that is not the focus of this paper.

3.3.1 Probability Smoothing: 391 Calculations

One problem with the above procedure is that the training set might not cover all of the possible values that might appear in the target's password. For example, the target's password might be "zebra789", but "789" may have not appeared in the training set. And ideally, an attacker wants to include these cases in their guesses by assigning them appropriate non-zero probability. One method we can find the proper probability is to use a variant of Laplacian smoothing:

$$P(x) = \frac{\text{number of times seen in the training set} + \alpha}{\text{total number of values} + \text{number of categories} * \alpha}$$

where α can be any real numbers between 0 and 1, where 0 represents no probability smoothing and 1 represents a large degree of probability smoothing. For example, suppose 500 two-digit numbers appeared in the training set, but the number 22 was not encountered. There are 100 categories of digit variables(for example, D_1 is one category, D_1 and D_2 are two categories). If the attacker chooses to smooth with $\alpha = 1$, then the probability of observing the number 22 would be computed as:

$$P(22) = \frac{0+1}{500+100*1} = 0.16\%$$

One way to interpret the smoothing formula is from a Bayesian point of view. This actually corresponds to the **expectation** of the **posterior distribution** for a symmetric **Dirichlet distribution** with α as a **prior distribution**. In class, we learned posterior and prior as two constant

numbers that represent the probability of a random variable H given another random variable E, and the probability of H before given E respectively. However, in a more generalized case, we are not often so sure about what the exact value should the posterior and the prior be. So we treat them as two random variables, for which their output values represent probabilities, with known probability distributions. Therefore, the expectation of the posterior distribution is a probability, which fits into the context of our smoothing formula.

The dirichlet distribution is a family of continuous multivariate probability distribution with parameter vector α. In Bayesian statistics, it is commonly used to describe the prior distributions. So in our case, it means that the prior distribution is a dirichlet distribution with parameter α.

A more intuitive understanding is seeing α as a **pseudocount**. It can be interpreted as the amount added to the number of observed cases in order to change the expectation in a model of those data when the expected probability is not known to be zero. In another words, a pseudocount α weighs the posterior distribution similarly to each category having an additional count of α. If the number of occurrence of i is x_i, and the size of the sample is N, the empirical probability of event i is

$$P_{\text{empirical}}(i) = \frac{x_i}{N}$$

the posterior probability when additively smooths by increasing each count x_i with α a priori is then

$$P_{\text{empirical}}(i) = \frac{x_i + \alpha}{N + \alpha d}$$

where d is the number of categories.

3.3.2 Generating the "Next" Function

After building the probabilistic context free grammar, the next step is to design an algorithm that uses this grammar to generate guesses in order of decreasing probability with optimal running time and memory space. In other words, we would like to generate the pre-terminal structures from highest probability to lowest probability, since dictionary words will be filled in to obtain the final terminal structure.

To optimize the running time of the algorithm, it should be designed to operate in an online mode. That is to say, it should first calculate the current best pre-terminal structure and output it to another distributed password cracker to generate terminal structures and test the guess. Then we update the current best pre-terminal structure to be another pre-terminal structure with the next highest probability, etc.

An example to illustrate the generating order of the pre-terminal structures for the grammar in Figure 6 is showed in Figure 7. The pre-structure with the highest probability 0.188 is node 1 "$4L_3$ $$4", so the algorithm would like to first output node 1 to the distributed password cracker. Then it would output node 2 "$L_3$4!" since 0.097 is the next highest probability.

One approach to implement this idea is to simply output all possible pre-terminal structures with their probability and sort them in decreasing probability order. However, it resulted in over a hundred gigabytes of data needed to store before actually guessing the password.

The actual solution is based on a data structure of a standard priority queue. We denote the position in which the variables appear by the *index* of a variable. For example, for a base structure $L_3D_1S_1$, the variable L_3 has index 0 and the variable D_1 has index 1. Then we order the substituted terminal value(such as 4 for D_1) in the pre-terminal structure in priority order (i.e. decreasing probability) for their respective class. So that when the distributed password cracker fills in the alpha variables with alphabets, they can quickly find the next alphabets to substitute. We also use a pivot value to ensure that all possible pre-terminal structures under the same base structure are

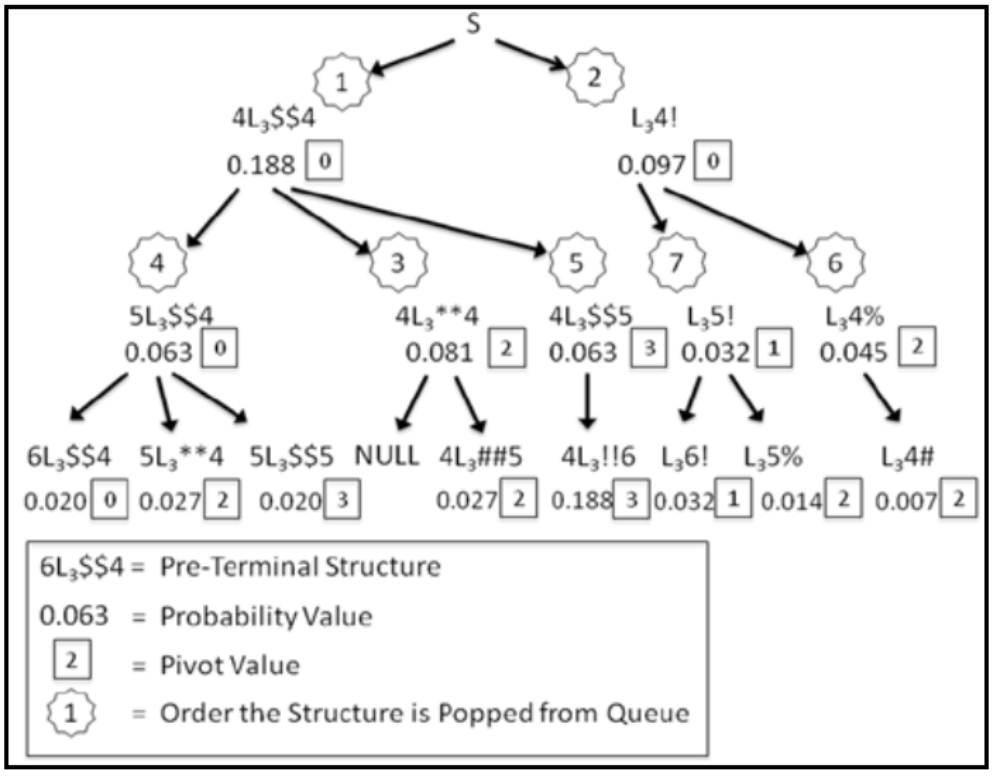

Figure 7: Generating the "Next" pre-terminal structures for the base structures in Figure 6

put into the priority queue without duplication. The pivot value is set to be the index changed when a pre-terminal structure is popped from the priority queue. A central password cracker first calculates the pre-terminal structures with decreasing probability order, and then each pre-terminal structure is fed in a distributed password cracker to get terminals by substituting alpha variables with words in dictionaries. The reason to have a central password cracker and distributed password crackers of a computation-efficiency concern.

To be clear of how this works, let's take a look at an example based on figure 7. Initially, the highest probability pre-terminals for every base structure is inserted into the priority queue with pivot value 0 (Figure 8) Next, the top entry in the priority queue is popped and passed on to a

Base Struct	Pre-Terminal	Probability	Pivot Value
$D_1L_3S_2D_1$	$4L_3\$\$4$	0.188	0
$L_3D_1S_1$	$L_34!$	0.097	0

Figure 8: Initial Priority Queue

distributed password cracker to continue generating terminal structures. Then from the rest of the pre-terminal strings with the form of $D_1L_3S_2D_1$, we find the strings that can be derived from the popped base structure by substituting terminals by terminals with next highest probability. In this case, since the popped pivot value was 0, all index variables could be substituted.So for variable index 0, 2, 3, the pre-terminals with the highest probability are inserted in the queue, and the

pivot value is set to be equal to the index that is changed (Figure 9). Note that once a new entry is inserted in the queue, the queue will reorder the entries based on their probability so that the entry with larger probability outout first.

Base Struct	Pre-Terminal	Probability	Pivot Value
$L_3D_1S_1$	$L_34!$	0.097	0
$D_1L_3S_2D_1$	$4L_3$**4	0.081	2
$D_1L_3S_2D_1$	$5L_3$$4	0.063	0
$D_1L_3S_2D_1$	$4L_3$$5	0.063	3

Figure 9: Priority Queue after the First Entry was Popped

Note that after the second entry in Figure 9 is popped, no new entry will be inserted because of its D_1 and S_2 variable. For D_1, this is because the index of D_1 is less than the pivot value. For S_2, if we look at the probability for S_2 listed in figure 6, we will find that "**" is already the terminal with the least probability. The process continues until the priority queue becomes empty, or the password has been cracked successfully.

Note that this algorithm is guaranteed to output the pre-terminal structures in decreasing order. To prove this statement, we first assume for the sake of contradiction that the statement does not hold. i.e, at some step of processing, there exist an entry x with probability that is strictly higher than a previously output entry y, or:

$$P(x) > P(y) \text{ and y is removed and output before x}$$

First we assert that x had a parent entry z (a parent entry of x means an entry that is popped one step before x and has the same base structure as x). If x has no parent, then x must be a highest probability pre-terminal structure for a base structure. Based on our assumption, y is removed before x and y has probability less than x. However, this is bizarre given how we construct the algorithm for its initialization phase. Thus, x must has a parent z.

Without loss of generality, we can assume that x is the first value produced by the algorithm that satisfies our assumption. It means that when z was output before x, it must not satisfy our assumption given the following relation:

$$P(z) >= P(x) > P(y)$$

Hence z must be output before y because it has a higher probability. Once z is output from the priority queue, x is inserted. This means that x was inserted in the priority queue before y has been removed. But with how we construct the queue data structure, it is impossible. This contradicts our assumption that y is popped before x. Therefore, our algorithm ensures to output the pre-terminal structures in non-increasing order. Another way to see this work is just by the property of queue. Since the entries in queue are always reordered according to their probabilities, this statement automatically becomes true. This property of queue in turn justifies why we employed this data structure instead of others.

4 Performance Comparison

In Charles Matthew Weir's thesis, he compared the performance of his software "UnLock" which implemented the probabilistic context free with the most popular password cracking software "John the Ripper" which employed all the traditional techniques we discussed in this paper. The performance of UnLock was significantly better than John the Ripper in all test sessions. As Figure 10 depicts, the PCFG method is able to crack more passwords than two different modes of John the Ripper when restricted to 300 million guesses.

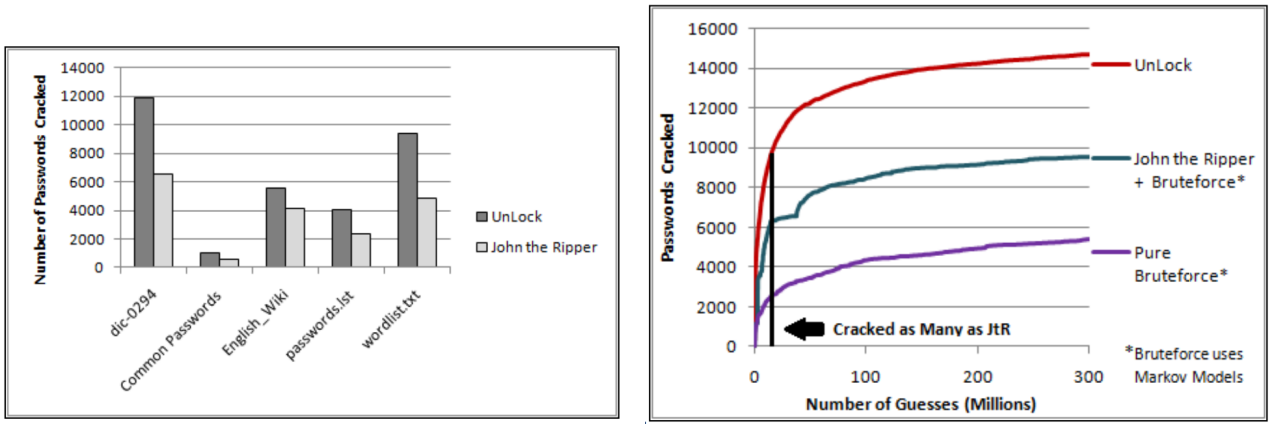

Figure 10: Performance Comparison of PCFG with John the Ripper. Trained on the MySpace Training List

Dürmuth's et al compared the performance of the Markov method with OMEN algorithm with other methods using three different publicly available password lists: the RockYou list (RY), MySpace list (MS), and the Facebook list (FB). All models were trained on a training subset from the RockYou list (RY-t), and were tested on the RockYou test list (RY-e) and the other two lists. The JtR-Markov implements the Markov method with the original password generation algorithm that doesn't generate in order of decreasing probability. It also compares with the PCFG password guesser of Weir at al. Interestingly, the OMEN Markov method outperforms all publicly available password guessers, including the PCFG method (Figures 11 and 12).

5 Conclusion

We covered the two major offline password attacking methods: brute force attacks and dictionary based attacks. Under brute force attacks, we introduced pure brute force attack, pure brute force attack with letter frequency analysis, and whole-string Markov method in detail. Under dictionary based attacks, we discussed a novel dictionary based creaking method with the model of probabilistic context-free grammars.

As pointed out by Charles Matthew Weir in the first sentence of his PhD thesis, "at its heart, a password cracking attack is just a guessing attack", the essence of all the techniques discussed in our report is how to make guesses. Since probability is about formalizing the procedures of making guesses, offline password cracking is all about using probability implicitly.

The success of the Probabilistic Context-Free Grammar method and Whole-string Markov method illustrates the importance of incorporating advanced modeling techniques in offline pass-

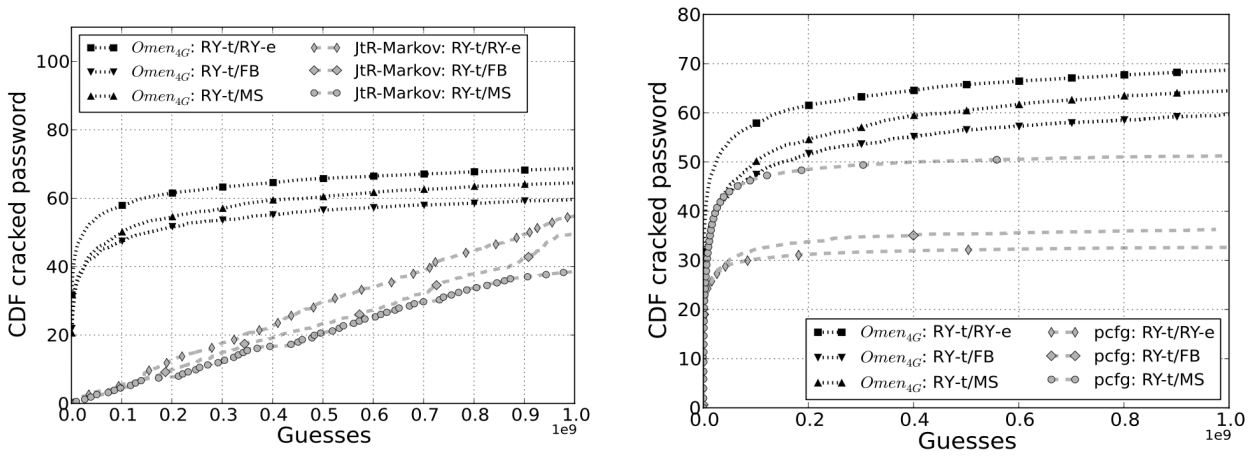

Figure 11: Performance Comparison of OMEN with John the Ripper Markov mode and PCFG at 1 billion guesses.

Algorithm	Training Set	#guesses	Testing Set		
			RY-e	MS	FB
Omen	RY-t	10 billion	80.40%	77.06%	66.75%
	RY-t	1 billion	68.7%	64.50%	59.67%
PCFG [24]	RY-t	1 billion	32.63%	51.25%	36.4%
JtR-Markov [16]	RY-t	10 billion	64%	53.19%	61%
	RY-t	1 billion	54.77%	38.57%	49.47%

Figure 12: Caption

word cracking. Going forward, with formulating the problem of password cracking more in-line with probability language and mathematics, there will be more chances that we can adopt more advanced modeling techniques that increasing the effectiveness of the password cracking techniques. On human users' side, password crackers can help inform better password strength estimators and guide users to pick stronger passwords. Since password cracking techniques are based on the assumption of human's tendencies and habits of creating passwords, this improvement of the security of passwords relies on asking users to not follow the expected tendencies. However, this can be hard, as passwords that are recognized to be stronger may also be harder to memorize. That being said, the key to the future of human memorable passwords is to enhance the underlying security in how passwords are stored and accessed so that offline password cracking became more and more impossible, instead of relying on people creating random generated password that is hard to trace a pattern.

References

[1] Claude Castelluccia, Markus Dürmuth, and Daniele Perito. Adaptive password-strength meters from markov models. In *NDSS*, 2012.

[2] Markus Dürmuth, Fabian Angelstorf, Claude Castelluccia, Daniele Perito, and Abdelberi Chaabane. Omen: Faster password guessing using an ordered markov enumerator. In *International symposium on engineering secure software and systems*, pages 119–132. Springer, 2015.

[3] Charles Matthew Weir. *Using probabilistic techniques to aid in password cracking attacks.* The Florida State University, 2010.

[4] Matt Weir, Sudhir Aggarwal, Breno De Medeiros, and Bill Glodek. Password cracking using probabilistic context-free grammars. In *2009 30th IEEE Symposium on Security and Privacy*, pages 391–405. IEEE, 2009.