2009 30th IEEE Symposium on Security and Privacy

Password Cracking Using Probabilistic Context-Free Grammars

Matt Weir, Sudhir Aggarwal, Breno de Medeiros, Bill Glodek
Department of Computer Science,
Florida State University, Tallahassee, Florida 32306, USA
weir@cs.fsu.edu, sudhir@cs.fsu.edu, breno.demedeiros@gmail.com, wjglodek@gmail.com

Abstract — **Choosing the most effective word-mangling rules to use when performing a dictionary-based password cracking attack can be a difficult task. In this paper we discuss a new method that generates password structures in highest probability order. We first automatically create a probabilistic context-free grammar based upon a training set of previously disclosed passwords. This grammar then allows us to generate word-mangling rules, and from them, password guesses to be used in password cracking. We will also show that this approach seems to provide a more effective way to crack passwords as compared to traditional methods by testing our tools and techniques on real password sets. In one series of experiments, training on a set of disclosed passwords, our approach was able to crack 28% to 129% more passwords than John the Ripper, a publicly available standard password cracking program.**

***Index Terms* — Computer security, Data security, Computer crime**

1. INTRODUCTION

Human-memorable passwords remain a common form of access control to data and computational resources. This is largely driven by the fact that human memorable passwords do not require additional hardware, be it smartcards, key fobs, or storage to hold private/public key pairs.

Trends that increase password resilience, in particular against off-line attacks, include current or proposed password hashes that involve salting or similar techniques [1]. Additionally, users are often made to comply with stringent password creation policies. While user education efforts can improve the chances that users will choose safer and more memorable passwords [2], systems that allow users to choose their own passwords are typically vulnerable to space-reduction attacks that can break passwords considerably more easily than through a brute-force attack (for a survey, see [3]).

Manuscript received November 10, 2008.
This work was supported in part by the U.S. National Institute of Justice under Grant 2006-DN-BX-K007.
M. Weir is a PhD student in the Computer Science Department at Florida State University; weir@cs.fsu.edu.
S. Aggarwal is Professor of Computer Science at Florida State University; sudhir@cs.fsu.edu.
B. de Medeiros works for Google and is a courtesy professor of Computer Science at Florida State University; breno.demedeiros@gmail.com.
B. Glodek graduated with an M.S. degree in Computer Science from Florida State University; wjglodek@gmail.com.

To estimate the risk of password-guessing attacks, it has been proposed that administrators pro-actively attempt to crack passwords in their systems [4]. Clearly, the accuracy of such estimates depends on being able to approximate the most efficient tools available to adversaries. Therefore, it is an established practice among security researchers to investigate and communicate advances in password-breaking: If the most efficient attack is indeed publicly known, then at least legitimate system operators will not underestimate the risk of password compromise. Moreover, password breaking mechanisms may also be used for data recovery purposes. This often becomes necessary when important data is stored in encrypted form under a password-wrapped key and the password is forgotten or otherwise unavailable. In this paper we describe novel advancements in password-breaking attacks.

Some improvements in password retrieval are achieved by increasing the speed with which the attacker can make guesses, often by utilizing specialty hardware or distributed computing [5, 6]. While increasing the speed at which you can make guesses is important, our focus is to try and reduce the number of guesses required to crack a password, and thus to optimize the time to find a password given whatever resources are available.

Our approach is probabilistic, and incorporates available information about the probability distribution of user passwords. This information is used to generate password patterns (which we call structures) in order of decreasing probability. These structures can be either password guesses themselves or, effectively, word-mangling templates that can later be filled in using dictionary words. As far as we are aware, our work is the first that utilizes large lists of actual passwords as training data to automatically derive these structures.

We use probabilistic context-free grammars to model the derivation of these structures from a training set of passwords. In one series of experiments, we first trained our password cracker on a training set of disclosed passwords. We then tested our approach on a different test set of disclosed passwords and compared our results with John the Ripper [11], a publicly available password cracking program. Using several different dictionaries, and allowing the same number of guesses, our approach was able to crack 28% to 129% more passwords than John the Ripper. Other experiments also showed similar results.

By training our attacks on known passwords, this approach also provides us a great deal of flexibility in tailoring our attacks since we automatically generate probability-valued structures from training data. For instance, we can train our password cracker on known Finnish passwords if our target is a native Finnish speaker.

2. BACKGROUND AND PREVIOUS WORK

In off-line password recovery, the attacker typically possesses only a hash of the original password. To crack it, the attacker makes a guess as to the value of the original password. The attacker then hashes that guess using the appropriate password-hashing algorithm and compares the two hashes. If the two hashes match, the attacker has discovered the original password, or in the case of a poor password hashing algorithm, they at least have a password that will grant them access.

The two most commonly used methods to make these guesses are brute-force and dictionary attacks. With brute-force, the attacker attempts to try all possible password combinations. While this attack is guaranteed to recover the password if the attacker manages to brute-force the entire password space, it often is not feasible due to time and equipment constraints. If no salting is used, brute-force attacks can be dramatically improved through the use of pre-computation and powerful time-memory trade-off techniques [7, 8].

The second main technique is a dictionary attack. The dictionary itself may be a collection of word lists that are believed to be common sources for users to choose mnemonic passwords [9]. However, users rarely use unmodified elements from such lists (for instance, because password creation policies prevent it), and instead modify the words in such a way that they can still recall them easily. In a dictionary attack, the attacker tries to reproduce this frequent approach to password choice, processing words from an input dictionary and systematically producing variants through the application of pre-selected mangling rules. For example, a word-mangling rule that adds the number "9" at the end of a dictionary word would create the guess, "password9", from the dictionary word "password". For a dictionary attack to be successful, it requires the original word to be in the attacker's input dictionary, and for the attacker to use the correct word-mangling rule. While a dictionary based attack is often faster than brute-force on average, attackers are still limited by the amount of word-mangling rules they can take advantage of due to time constraints. Such constraints become more acute as the sizes of the input dictionaries grow. In this case, it becomes important to select rules that provide a high degree of success while limiting the number of guesses required per dictionary word.

Choosing the right word-mangling rules is crucial as the application of each rule results in a large number of guesses. This is especially true when the rules are used in combination. For example, adding a two-digit number to the end of a dictionary word for a dictionary size of 800,000 words [9] would result in 80,000,000 guesses. Changing the first letter to be both uppercase and lowercase would double this figure. Furthermore, in a typical password retrieval attempt it is necessary to try many different mangling rules. The crucial question then becomes, which word-mangling rules should one try and in which order?

Narayanan and Shmatikov use Markov models to generate probable passwords that are phonetically similar to words and that thus may be candidates for guesses [10]. They further couple the Markov model with a finite state automaton to reduce the search space and eliminate low probability strings. The goal of their work, however, is to support rainbow-based pre-computation (and, subsequently very fast hash inversion) by quickly finding passwords from dictionaries that only include linguistically likely passwords. They thus do not consider standard dictionary attacks.

Our approach can be viewed as an improvement to the standard dictionary-based attack by using existing corpuses of leaked passwords to automatically derive word-mangling rules and then using these rules and the corpus to further derive password guesses in probability order. We are also able to derive more complex word-mangling rules without being overwhelmed by large dictionaries due to the assignments of probabilities to the structures.

3. PROBABILISTIC PASSWORD CRACKING

Our starting assumption is that not all guesses have the same probability of cracking a password. For example, the guess "password12" may be more probable than the guess "P@$$W0rd!23" depending on the password creation policy and user creativity. Our goal is thus to generate guesses in decreasing order of probability to maximize the likelihood of cracking the target passwords within a limited number of guesses.

The question then becomes, "How should we calculate these probabilities?" To this end, we have been analyzing disclosed password lists. These lists contain real user passwords that were accidentally disclosed to the public. Even though these passwords are publicly available, we realize they contain personal information and thus treat them as confidential.

For our experiments we needed to divide the password lists up into two parts, a training corpus and a test corpus. If a password appears in our training corpus, we will not use it in the test corpus. In the case of password lists that were disclosed in plain-text format, (i.e. prior to any hashing), we can choose to use the passwords in either the training or the test corpuses. If a list of password hashes was instead disclosed, we used the entire list in the test corpus. This is because we have to crack the password hashes before we can know what the plain text words were that created them. By separating the training and test corpuses we can then compare the effectiveness of our probabilistic password cracking with other publicly available password cracking attacks, notably John the Ripper [11], by comparing their results on the test sets.

3.1 PREPROCESSING

In the preprocessing phase, we measure the frequencies of certain patterns associated to the password strings. First we define some terminology that is used in the rest of the paper.

Let an *alpha string* be a sequence of alphabet symbols. Also let a *digit string* be a sequence of digits, and a *special string* be a sequence of non-alpha and non-digit symbols. When parsing the training set, we denote alpha strings as **L**, digit strings as **D**, and special strings as **S**. For example the password "$password123" would define the simple structure **SLD**. The base structure is defined similarly but also captures the length of the observed substrings. In the example this would be $\mathbf{S_1L_8D_3}$. See Table 3.1.1 and Table 3.1.2. Note that the character set for alpha strings can be language dependent and that we currently do not make a distinction between upper case and lower case.

The first preprocessing step is to automatically derive all the observed base structures of all the passwords in the training set and their associated probabilities of occurrence.

TABLE 3.1.1
Listing of different string types

Data Type	**Symbols**	**Examples**
Alpha String	abcdefghijklmnopqrstuvwxyzäö	cat
Digit String	0123456789	432
Special String	!@#$%^&*()-_=+[]{};':",./<>?	!!

TABLE 3.1.2
Listing of different grammar structures

Structure	**Example**
Simple	SLD
Base	$S_1L_8D_3$
Pre-Terminal	$$L_8$123
Terminal (Guess)	$wordpass123

For example, the base structure $\mathbf{S_1L_8D_3}$ might have occurred with probability 0.1 in the training set. We decided to use the base structure directly in our grammars rather than the simple structure since the derivation of the base structure from the simple structure was unlikely to be context-free.

The second type of information that we obtained from the training set was the probability of digit strings and of special strings appearing in the training set. To see an example of this please refer to Table 3.1.3 and Table 3.1.4.

TABLE 3.1.3
Probabilities of one-digit numbers

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

TABLE 3.1.4
Probabilities of top 10 two-digit numbers

2 Digits	**Number of Occurrences**	**Percentage of Total**
12	1084	5.99425
13	771	4.26344
11	747	4.13072
69	734	4.05884
06	595	3.2902
22	567	3.13537
21	538	2.97501
23	533	3.94736
14	481	2.65981
10	467	2.58239

We choose to calculate the probabilities only for digit strings and special strings since we knew that the corpus of words, (aka alpha strings), that users may use in password generation was much larger than what we could accurately learn from the training set. Note that the calculation of the digit string and special string probabilities is gathered independently from the base structures in which they appear.

Please also note that all the information that we capture of both types is done automatically from an input file of training passwords, using a program that we developed.

Referring to Table 3.1.2 again, the *pre-terminal structure* fills in specific values for the **D** and **S** parts of the base structure. Finally, the *terminal structure* fills in a specific set of alphabet letters for the **L** parts of the pre-terminal structure. Deriving these structures is discussed next.

3.2 USING PROBABILISTIC GRAMMARS

Context-free grammars have long been used in the study of natural languages [12, 13, 14], where they are used to generate (or parse) strings with particular structures. We show in the following that the same approach is useful in the automatic generation of password guesses that resemble human-created passwords.

A context-free grammar is a defined as G = (**V, Σ, *S*, P**), where: **V** is a finite set *of variables* (or non-terminals), **Σ** is a finite set of *terminals*, ***S*** is the *start variable*, and **P** is a finite set of *productions* of the form (1):

$$\boldsymbol{\alpha \rightarrow \beta} \qquad (1)$$

where **α** is a single variable and **β** is a string consisting of variables or terminals. The language of the grammar is the set of strings consisting of all terminals derivable from the start symbol.

Probabilistic context-free grammars simply have probabilities associated with each production such that for a specific left-hand side (LHS) variable all the associated productions add up to 1. From our training set, we first derive a set of productions that generate the base structures and another set of productions that derive terminals consisting of digits and special characters. In our grammars, in addition to the start symbol, we only use variables of the form $\mathbf{L_n}$,$\mathbf{D_n}$, and $\mathbf{S_n}$, for specified values of *n*. We call these variables *alpha variables*, *digit variables* and *special variables* respectively. Note that rewriting of alpha variables is done using an input dictionary similar to that used in a traditional dictionary attack.

A string derived from the start symbol is called a *sentential form* (it may contain variables and terminals). The probability of a sentential form is simply the product of the probabilities of the productions used in its derivation. In our production rules, we do not have any rules that rewrite alpha variables; thus we can "maximally" derive sentential forms and their probabilities that consist of terminal digits, special characters and alpha variables. These sentential forms are the pre-terminal structures.

In our preprocessing phase, we automatically derive a probabilistic context-free grammar from the training set. An example of such a grammar is shown in Table 3.2.1. Given this grammar, we can furthermore derive, for example, the pre-terminal structure:

$$\boldsymbol{S \rightarrow L_3D_1S_1 \rightarrow L_34S_1 \rightarrow L_34!} \qquad (2)$$

with associated probability of 0.0975. The idea is that pre-terminal structures define mangling rules that can be directly used in a distributed password cracking trial. For example, a control server could compute the pre-terminal structures in order of decreasing probability and pass them to a distributed system to fill in the dictionary words and hash the guesses. The ability to distribute the work is a major requirement if the proposed method is to be competitive with existing alternatives. Note that we only need to store the probabilistic context-free grammar and that we can derive the pre-terminal structures as needed. Furthermore, note that fairly complex base structures might occur in the training data and would eventually be used in guesses, but the number of base structures is unlikely to be overwhelming.

TABLE 3.2.1
Example probabilistic context-free grammar

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

The order in which pre-terminal structures are derived is discussed in Section 3.3. Given a pre-terminal structure, a dictionary is used to derive a terminal structure which is the password guess. Thus if you had a dictionary that contained {cat, hat, stuff, monkey} the previous pre-terminal structure $L_3$4! would generate the following two guesses (the terminal structures), {cat4!, hat4!}, since those are the only dictionary words of length three.

There are many approaches that could be followed when substituting the dictionary words in the pre-terminal structures. Note that each pre-terminal structure has an associated probability.

One approach to generating the terminal structures is to simply fill in all relevant dictionary words for the highest probability pre-terminal structure, and then choose the next highest pre-terminal structure, etc. This approach does not further assign probabilities to the dictionary words. The naturalness of considering this approach is that we are leaning only lengths of alpha strings but not specific replacements from the training set. This approach thus always uses pre-terminal structures in highest probability order regardless of the input dictionary used. We call this approach *pre-terminal probability order*.

Another approach is to assign probabilities to alpha strings in various ways. Without more information on the likelihood of individual words, the most obvious technique is to assign the alpha strings a probability based on how many words of that length appear in the dictionary. If there are 10 words of length 3, then the probabilities of each of those words would be 0.10. We call this approach *terminal probability order*. Note that in this case each terminal structure (password guess) has a well-defined probability. The probability however is based in part on the input dictionary which was not learned during the training phase. We also considered other approaches for assigning probabilities to alpha strings. For instance it is possible to assign probabilities to words in the dictionary based on other criteria such as observed use, frequency of appearance in the language, or knowledge about the target.

An approach related to pre-terminal probability order is to use the probability of the pre-terminals to sample a pre-terminal structure and then fill in appropriate dictionary words for the alpha strings. Notice that in this latter case, we would not use a pre-terminal necessarily in highest probability order, but the frequency of generating terminals over time would match this pre-terminal probability. We call this approach *pre-terminal sampled order*.

In this paper, we will only consider results using pre-terminal probability order and terminal probability order. We remark that the terminal order uses the *joint probability* determined by treating the probabilities of pre-terminal structures and of the dictionary words that are substituted in as independent.

It should be noted that we use probabilistic context-free grammars for modeling convenience only; since our production rules derived from the training set do not have any recursion, they could also be viewed as regular grammars. In fact, this allows us to develop an efficient algorithm to find an indexing function for the pre-terminal structures, as discussed in the next section. The grammars that we currently automatically generate are unambiguous context-free grammars.

3.3 EFFICIENTLY GENERATING A "NEXT" FUNCTION

In this section we consider the problem of generating guesses in order of decreasing (or equal) probability and describe the algorithm. For pre-terminal probability order, this means in decreasing order of the pre-terminal structures. For terminal probability order, this is the probability of the terminal structures. However, the "next" function algorithm is the same in both cases except that for the terminal probability order, the initial assignment of probabilities to the starting pre-terminal structures includes the probabilities of the alpha variables. In Section 3.4, we outline the proof of correctness of the algorithm.

First note that it is trivial to generate the most probable guess. One simply replaces all the base structures with their highest probability terminals and then selects the pre-terminal structure with the highest probability. Note that for terminal probability order, the alpha strings in the base structure are also assigned a probability. For example, using the data in Table 3.2.1, the highest probable pre-terminal structure would be $4L_3$$4. Since there are only 1589 base structures generated by our largest training set, this is not difficult. However, a more structured approach is needed to generate guesses of a rank other than the first.

To optimize the total running time of the algorithm, it is useful if it can operate in an online mode, i.e. it calculates the current best pre-terminal structure and outputs it to the underlying (also distributable) password cracker. On the other hand, also for performance reasons, at any particular stage the algorithm should only calculate those pre-terminal structures that might be the current most probable structure remaining, taking into consideration the last output value. Referring to Fig. 3.3.1, we would like to generate the pre-terminal structures $L_3$5! and $L_3$4% (nodes 7 and 6) only after $L_3$4! (node 2) has been generated.

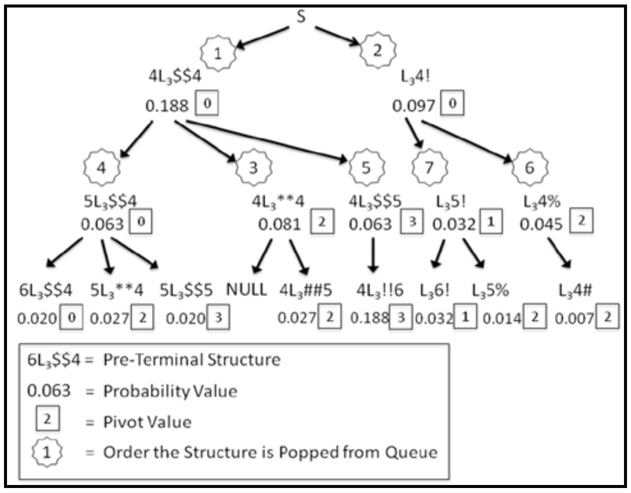

Fig. 3.3.1. Generating the "Next" Pre-terminal Structures for the Base Structures in Table 3.2.1 (partial tree shown).

One approach that is simple to describe and implement is to output all possible pre-terminal structures, evaluate the probability of each, and then sort the result. Unfortunately this pre-computation step is not parallelizable with the password cracking step that follows (i.e., it is not an online algorithm).

Originally when we were still trying to see if using probabilistic grammars was worth further investigation, we created a proof of concept program that took this approach. Unfortunately in addition to the problems described above, it also resulted in over a hundred gigabytes of data that we had to generate and then sort before we could make our first password guess. As you can imagine, this does not lend itself to a real world application.

Our actual solution adopts as its main data structure a standard priority queue, where the top entry contains the most probable pre-terminal structure. In the following, we denote by the *index* of a variable in a base structure to mean the position in which the variable appears. For example, in the base structure $L_3D_1S_1$ the variable L_3 would be assigned an index of 0, D_1 an index of 1, and S_1 an index of 2. Next, we order all terminal values, (such as the numbers 4, and 5 for D_1) in priority order for their respective class. That way we can quickly find the next most probable terminal value.

The structure of entries in the priority queue can be seen in Table 3.3.2. They contain a base structure, a pre-terminal structure, and a pivot value. This *pivot value* is checked when a pre-terminal structure is popped from the priority queue. The pivot value helps determine which new pre-terminal structures may be inserted into the priority queue next. The goal of using pivot values is to ensure that all possible pre-terminal structures corresponding to a base structure are put into the priority queue without duplication.

More precisely, the pivot value indicates that the pre-terminal structures to be next created from the original base structure are to be obtained by replacing variables with an *index value* equal to or greater than the popped pivot value. Let's look at an example based on the data in Table 3.2.1. Initially all the highest probability pre-terminals from every base structure will be inserted into the priority queue with a pivot value of 0. See Figure 3.3.1 and Table 3.3.2.

TABLE 3.3.2
Initial Priority Queue

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

Next, the top entry in the priority queue will be popped. The pivot value will be consulted, and child pre-terminal structures will be inserted as part of new entries for the priority queue. These pre-terminal structures are generated by substituting variables in the popped base structure by values with next-highest probability. Note that only one variable is replaced to create each new candidate entry. Moreover, this replacement is performed (as described above) for each variable with index equal to or greater than the popped pivot value. The new pivot value assigned to each inserted pre-terminal structure is equal to the index value of the variable that was substituted. See Fig. 3.3.1 and Table 3.3.3 to see the result after popping the top queue entry. Also see Appendix 1.

TABLE 3.3.3
Priority queue after the first entry was popped

Base Struct	**Pre-Terminal**	**Probability**	**Pivot Value**
$L_3D_1S_1$	$L_3$4!	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

In this instance, since the popped pivot value was 0, all index variables could be substituted. L_3 was not incremented since there were no values to fill in for it, as the alpha strings are handled by the password cracker in a later stage. Both of the D_1 structures and S_2 were replaced, resulting in three new pre-terminal structures being inserted into the queue with pivot values of 0, 2 and 3. Notice that when the priority queue entry corresponding to the 2rd row of Table 3.3.3 is popped, it will not cause a new entry to be inserted into the priority queue for its first D_1 or its S_2 structure. This is because $4L_3$**4's pivot value is equal to 2, which means that it cannot replace the first D_1 structure with an index value of 0. As for the S_2 structure, since '**' is the least probable terminal variable, there is no next-highest replacement rule and this entry will simply be consumed.

Observe that the algorithm is guaranteed to terminate because it processes existing entries by removing them and replacing them with new ones that either (a) have a higher value for the pivot or (b) replace the base structure variable in the position indicated by the pivot by a terminal that has lower probability than the current terminal in that position. It can moreover be easily ascertained that the pre-terminal structures in the popped entries are assigned non-increasing probabilities and therefore the algorithm can output these structures for immediate use as a mangling rule for the underlying distributed password cracker.

This process continues until no new pre-terminal structures remain in the priority queue, or the password has been cracked. Note that we do not have to store pre-terminal structures once they are popped from the queue, which has the effect of limiting the size of the data structures used by the algorithm. In section 4.5, we discuss the space complexity of our algorithm in detail in the context of our experimental results.

The running time for the current implementation of our next algorithm for generating guesses is extremely competitive with existing password cracking techniques. On one of our lab computers, (MaxOSX 2.2GHz Intel Core 2 Duo) it took on average 33 seconds to generate 37781538 unhashed guesses using our method. Comparatively, the popular password cracking tool John the Ripper [11] operating in wordlist mode took 28 seconds to make the same number of guesses. If we expand the number of guesses to 300 million, our technique took on average 3 minutes and 23 seconds to complete, while John the Ripper operating in incremental (brute-force) mode took 2 minutes and 55 seconds. Note that the vast majority of time (often weeks) taken in cracking passwords is spent in generating the hashes from those guesses and not in the generation of the actual guesses themselves. Because of this, even an extra minute or two spent generating guesses would be minor, and thus the running times of these two methods are essentially identical.

3.4 PROOF OF CORRECTNESS OF THE NEXT FUNCTION

Property P1: pre-terminal structures are output in non-increasing probability order.

Proof that *P1* holds:

1. Remember that the priority queue is initialized with one entry per base structure, and that the entry contains the pre-terminal structure with maximum probability for that base structure. These entries can be easily constructed by simply replacing the highest likelihood terminal values for all the non-alpha variables in each base structure.
2. Remember that the processing of an entry in the priority queue results in its removal and output, and (possibly) in the insertion of new entries. For convenience of description, we call these new entries "the children" and the removed entry "the parent". Recall that children never contain pre-terminal structures of strictly higher probability than the pre-terminal structure contained in the parent.

For the sake of contradiction, assume that *P1* does not hold, i.e., that at some step of processing, an entry x is output of strictly higher probability than a previously output entry y. That is:

$Prob(x) > Prob(y)$ and y is removed and output before x.

First let's argue that x had a parent entry z. Indeed, if x has no parent, then it was inserted in the priority queue during the algorithm initialization (when the highest probability pre-terminal structure for each base structure was inserted). But that means that x was in the priority queue at the step where y was output, in violation of the priority queue property. This contradiction implies that x had a parent z.

Without loss of generality, we can also assume that x is the first value produced by the algorithm that violates *P1*. Consequently, when z was output, it did not violate this property, and since:

$$Prob(z) >= Prob(x) > Prob(y),$$

it follows that z must have been output (and processed) before y. That means that x was inserted in the priority queue prior to y's removal, again in violation of the priority queue property. This final contradiction concludes the proof.

Note that by meeting the following conditions we can fully prove the required correctness of the next function:

- No duplicate pre-terminal structures are entered into the priority queue.
- All possible pre-terminal structures resulting from base structures are eventually entered into the priority queue.

Due to space requirements we do not include a proof of these conditions but it follows from our use of the pivot values.

4. EXPERIMENTS AND RESULTS

4.1 DESCRIPTION OF PASSWORD LISTS

For the research in this paper we obtained three password lists to try different cracking techniques against. All three lists represent real user passwords, which were compromised by hackers and subsequently publicly disclosed on the Internet. As stated before, we realize that while publicly available, these lists contain private data; therefore we treat all password lists as confidential. If you wish a copy of the list please contact the authors directly. Due to the moral and legal issues with distributing real user information, we will only provide the lists to legitimate researchers who agree to abide by accepted ethical standards.

The first list, hereafter referred to as the "MySpace List", was originally published in October 2006. The passwords were compromised by an attacker who created a fake MySpace login page and then performed a standard phishing attack against the users. The attacker did not secure the server they were collecting passwords upon which allowed independent security researchers to obtain copies of the passwords. One of these researchers, (not affiliated with any university), subsequently posted his copy of the list on the Full-Disclosure mailing list [15]. While multiple versions of the MySpace list exist, owing to the fact that different researchers downloaded the list at different times, we choose to use the version posted on Full-Disclosure which contained 67042 plain text passwords. Please note that not all of these passwords represent actual user passwords. This is because some users recognized that it was a phishing attack and entered fake, (and often vulgar), data. For our test we did not attempt to purge these fake passwords due to the difficulties in distinguishing between fake and real passwords.

The second list will be referred to as the SilentWhisper list. This list contains 7480 plain text passwords and was originally from the website www.silentwhisper.net. A hacker compromised the site via SQL injection, and due to a feud with the site owner, subsequently posted the list to bittorrent. As a special note, these later passwords were extremely basic. Only 3.28% of the passwords contained both letters and numbers, and only 1.20% of them contained both letters and special characters. A grand total of two of the passwords contained letters, numbers and special characters. We included this list though as it does represent the passwords many users choose.

The final list will be referred to as the “Finnish List”. This list was obtained by a hacker group via SQL injection attacks and the results were subsequently posted on the internet [16]. This list actually contains the passwords from many different sites that were compromised; most of them based in Finland, hence the name. This list contains 15699 passwords in plain text and an additional 22733 unique MD5 password hashes. It is important to note that the plain text passwords and the hashed passwords represent different user bases as they came from separate compromised sites. In fact, it appears that each set, (both the MD5 and plaintext lists), are composed of several lists from distinct websites that were broken into.

4.2 EXPERIMENT SETUP AND PROCEDURES

In the current implementation of our probabilistic password cracking guess generator, (written in C), our program is trained on an existing password list. Then once it is given an input dictionary it can generate password guesses based on either the pre-terminal probability or the terminal probability of the password structures. It is important to note that the training need only be done once to generate the grammar that will be used. This means that any group can create many different targeted grammars and then distribute them to the end users of the password cracking program. The end user would use input dictionaries of their choosing to crack passwords. Note that the storage requirement of a grammar is likely to be significantly less than the storage requirements of a typical input dictionary. Section 4.5 discusses space requirements in greater detail. This distinction between training and operation, and the small size of the base grammar generated means that our method is highly portable.

Our program currently outputs these guesses to stdout. This gives us the flexibility to use our guesses as input to various other password cracking programs. For instance, to test against a test set of plaintext passwords, we can simply check for an exact match, and record how many guesses were necessary before the first match could be found. For password lists that are hashed, such as the Finnish list, we piped the guesses generated by our program into the popular password cracking program John the Ripper [11]. Essentially this allows us to use our program’s word-mangling rules without having to code our own hash evaluator.

As a comparison against our probabilistic password cracking technique, we decided to use John the Ripper’s default word-mangling rules. These word-mangling rules are as close to an industry standard as we could find, and represent the approach most people would take when performing a dictionary-based password cracking attack. At its core, both our probabilistic password cracking guess generator and John the Ripper operating in wordlist mode are dictionary based attacks. When comparing the two methods, we ensure both programs use the same input dictionaries when trying to crack a given password set. In a dictionary-based attack, the number of guesses generated is finite, and determined by the size of the dictionary and the type of word-mangling rules used. To reflect this, unless otherwise specified, we limited the number of guesses our probabilistic password generator was allowed to create based on the number of guesses generated by the default John the Ripper rule set. This is because our program can generate many more rules than what is included in the default John the Ripper configuration and thus would create more guesses given the chance. By ensuring both methods are only allowed the same number of guesses, we feel we can fairly compare the two approaches.

To use our method, we have to train our password cracker on real passwords. To do this, we needed to separate our password lists into training lists and test lists. As a special note, if a password was used to train our method we made sure we did not include it in any of our test lists. We created two different training lists to train our probabilistic password cracker. The first list was created from the MySpace password list. We divided the MySpace password list into two parts, a training list and a test list. The MySpace training list contained a total of 33561 passwords. For the second training list, we used all of the plaintext passwords from the Finnish list. This contained a total of 15699 passwords. We used all the Finnish plaintext passwords since we used the Finnish hashed passwords for the test set. We did not create a training list from the SilentWhisper set due to its small size and the fact that we would need to have passwords left over to test against.

We then designated all passwords not in the training set as the test set. These passwords are never trained against, and are used solely to gauge the effectiveness of both our probabilistic password cracker and John the Ripper on real world passwords. Just as in standard machine learning research, our goal is by keeping these two groups, (training and testing), separate so we can avoid overtraining our method and provide a more accurate estimation of its potential. In summary, the three test lists we used were the MySpace test list containing 33481 plaintext passwords, the SilentWhisper list which contained 7480 plaintext passwords and the Finnish test list which contained 22733 unique MD5 password hashes.

One final note; a password was considered 'cracked' if the program generated a guess that matched the password in the test list.

4.3 DESCRIPTION OF INPUT DICTIONARIES

Due to the fact that both our password cracker and John the Ripper in wordlist mode operate as a dictionary attack, they both require an input dictionary to function. We choose a total of six publicly available input dictionaries to use in our tests. Four of them, "English_lower", "Finnish_lower", "Swedish_lower" and "Common_Passwords" were obtained from John the Ripper's public web site [11]. As a side note, the word "lower" refers to the fact that the dictionary words are stored as all lower case. Additionally we used the input dictionary "dic-0294" which we obtained from a popular password-cracking site [9]. This list was chosen due to the fact that we have found it very effective when used in traditional password crackers. Finally, we created our own wordlist "English_Wiki" which is based on the English words gathered off of www.wiktionary.org. This is a sister project of Wikipedia, and it provides user updated dictionaries in various languages.

Each dictionary contained a different number of dictionary words as seen in Table 4.3.1. Due to this, the number of guesses generated by each input dictionary when used with John the Ripper's default mangling rules also varied as can be seen by Fig. 4.3.2.

Table 4.3.1
Size of Input Dictionaries

Dictionary Name	Number of Dictionary Words
Dic-0294	869228
English_Lower	444678
Common_Passwords	816
English_Wiki	68611
Swedish_Lower	14555
Finnish_Lower	358963

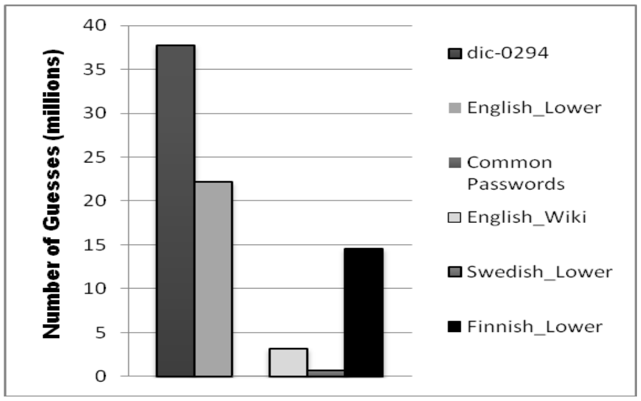

Fig. 4.3.2. Number of Password Guesses Generated by JtR

4.4 PASSWORD CRACKING RESULTS

Our first test, pictured in Fig. 4.4.1, shows the results of training our Probabilistic Password Cracker on the MySpace training list. Three different cracking techniques are used on the MySpace test list. The first is the default rule set for John the Ripper. The second technique is our Probabilistic Password Cracker using the pre-terminal probabilities of its structures. Once again, the pre-terminal probabilities do not assign a probability value to the dictionary words. The third technique is our Probabilistic Password Cracker using the probabilities of the terminals (guesses). Recall that in this case, we assign probabilities to dictionary words and extend our probabilistic context-free grammar to terminal strings. Once again, the number of guesses allowed to each run is shown in Fig. 4.3.2.

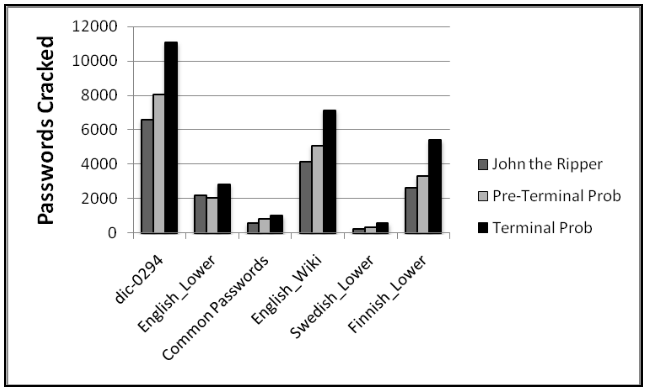

Fig. 4.4.1. Number of Passwords Cracked. Trained on the MySpace Training List. Tested on the MySpace Test List

As the data shows, our password cracking operating in terminal probability order performed the best. Using it, we achieved an improvement over John the Ripper ranging from

28% to 129% more passwords cracked given the same number of guesses. Additionally, when we used the pre-terminal order, in all cases but one we also achieved better results than John the Ripper, though less then what we achieved using terminal probability order.

The next question would be how does our probabilistic method work when trained on a different data set? The same test as above was run on the MySpace test list, but this time we used the Finnish training list to train our password cracker. The results are shown in Fig. 4.4.2.

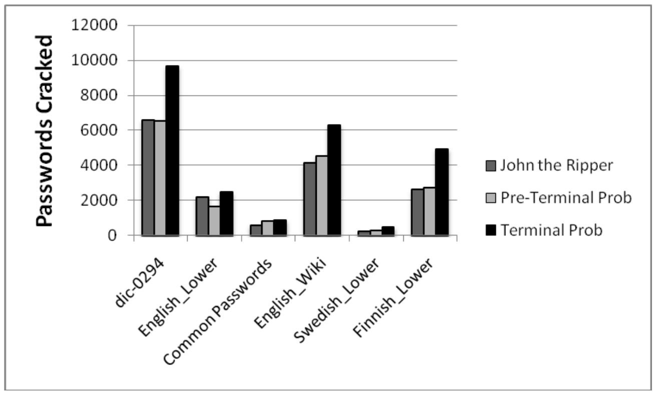

Fig. 4.4.2. Number of Passwords Cracked. Trained on the Finnish Training List. Tested on the MySpace Test List

As the results show, the terminal probability order once again performed the best, though not as well as it did when it was trained on the MySpace data. This time the improvement ranged from 11% to 96% more passwords cracked compared to John the Ripper. A surprising result to us was that when we used Pre-Terminal Probability Order, it did not result in a noticeable improvement over John the Ripper's default rule set. In fact, in two of the test cases it actually performed worse.

Next we ran the same tests by training our Probabilistic Password Cracker on the MySpace training list, and then running it against the SilentWhisper test list. The results can be seen in Fig. 4.4.3. As expected, in this case the default John the Ripper word-mangling rules performed slightly better. This is due to the relative simplicity of the SilentWhisper test set. Since our probabilistic method had been trained on more complex passwords, it spent much of its time generating guesses using advanced mangling rules, vs. John the Ripper which exhausted the simple mangling rules, (such as just use the dictionary word), first. This does show a limitation of our probabilistic method as it does need to be trained on passwords of similar complexity as the passwords it is trying to crack. That being said, in all of the test runs with the exception of the one using the English_Lower dictionary, our method operating in terminal probability order performed competitively with John the Ripper.

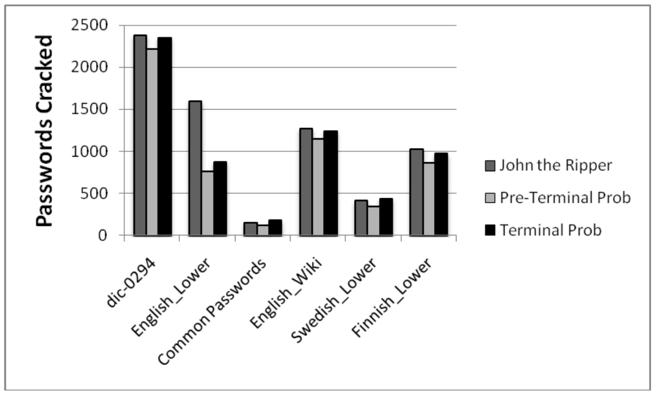

Fig. 4.4.3. Number of Passwords Cracked. Trained on the MySpace Training List. Tested on the SilentWhisper Test List

To round things out, we then evaluated our probabilistic method by training it on the Finnish training set and then attacking the Finnish test set. Please note that the Finnish training set and the Finnish test set were gathered from separate websites. Thus for this experiment, even though the users share a common language, we trained and then tested our password cracker against different user bases. Due to the time it takes to audit these passwords since they are hashed, we only performed this test with John the Ripper's default rule set and our method operating in terminal probability order. The results can be seen in Fig. 4.4.4.

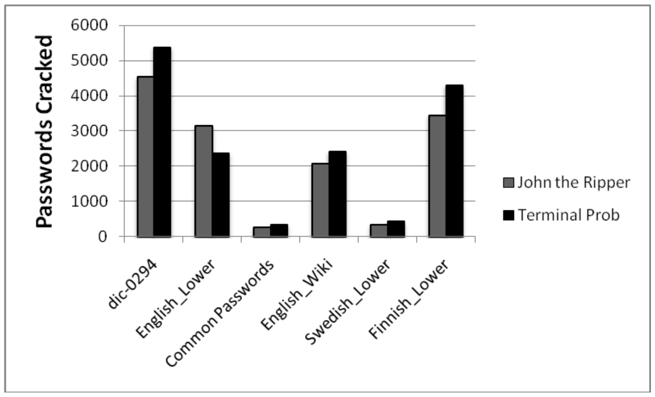

Fig. 4.4.4. Number of Passwords Cracked. Trained on the Finnish Training List. Tested on the Finnish Test List

While the results were not as dramatic as compared to cracking the MySpace list, we still see an improvement ranging from 17% to 29% over John the Ripper in all but one of the test cases. Considering that we had no previous knowledge of how the passwords in the test set compared in complexity to the passwords in the training set, this is still a fairly significant improvement. What this means is that we were able to train our password cracker on one user base and then use is successfully against another group which we knew nothing about except for their native language.

Looking back through the previous tests as shown in Fig. 4.4.1 through Fig. 4.4.4, one thing we noticed was that our probabilistic method performed significantly worse when it used the English_Lower dictionary compared to the results it obtained using the other input dictionaries. For example, let's consider the test, Fig 4.4.1, where we trained our attack on the MySpace training set, and tested it against the MySpace test set. If we exclude the run that used the English_Lower dictionary, the average improvement of our method using terminal probability order compared to John the Ripper was 90%. The improvement on the run which used the English_Lower dictionary was 28%. The other tests show similar results. We are still investigating why our attacks do not perform as well with this dictionary. Despite its name, the English_Lower dictionary seems to be comprised mostly of "made up" words, such as 'zoblotnick', and 'bohrh'. Our current assumption is that the presence of a large number of nonsense words throws off our method in two different ways. First our program wastes time trying these nonsense words. Second, when operating in terminal probability order, a large number of essentially "junk" words can make what should be a highly probable structure have a lower probability, and thus not be tried as soon. We still need to investigate this more thoroughly.

The next test we ran was to evaluate how the size of the training list affected our probabilistic password cracker. To investigate this we used training lists of various sizes selected from the original MySpace training list. The size of these lists is denoted by the number after them, aka the MySpace20K list contains twenty thousand training passwords. For reference, the MySpaceFull list contains all 33,561 training passwords from the MySpace training list.
We were concerned about sampling bias as the lists became shorter, (such as containing only 100 or 500 passwords). To address this, for all training sets containing less than one thousand passwords we trained and then ran each test twenty five times with a different random sample of passwords included in the training list each time. We then averaged the results of the 25 different runs. All the tests to measure the effect of the training list size used Terminal Probability Order and were run against the MySpace Test List. The results can be seen in Fig. 4.4.5. For comparison, John the Ripper's performance is the left-most value, and training sets increase in size from left to right for each input dictionary.

It was surprising that even when our password cracker was trained on only 10,000 passwords, our Probabilistic Method performed only slightly worse than when it was trained on 33,561 passwords. What was more surprising was that our password cracker performed comparable to John the Ripper even when it was trained on only 100 input passwords. We expect that given a longer run (aka allowing our password cracker to generate more guesses), the effect of having a larger training set will become more pronounced as it will generally provide the password cracker more base structures as well as digit and symbol strings to draw upon. Also, we expect that the larger training set would better reflect more accurate probabilities of the underlying base structures and replacement values.

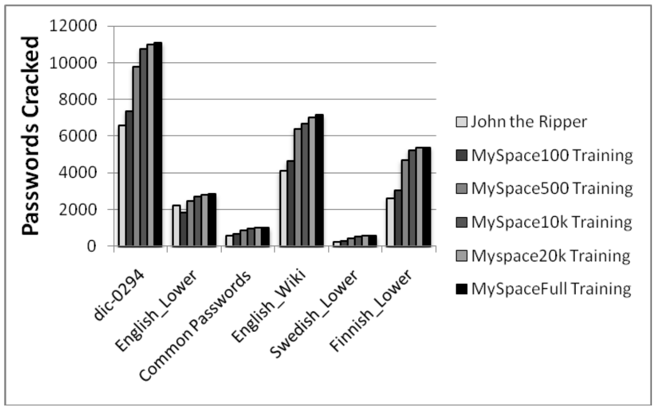

Fig. 4.4.5. Number of Passwords Cracked. Trained on different sized MySpace Training Lists. Tested on the MySpace Test List using Terminal Order Probability

In all the previous tests we limited our probabilistic method to the number of guesses generated by the default rule set of John the Ripper. One last test we wanted to run was to see how our probabilistic method performed if we let it continue to run over an extended time. The following Fig. 4.4.6 shows the number of passwords cracked over time using our probabilistic method operating in terminal probability order. Please note, while John the Ripper exited after making 37,781,538 guesses, we continued to let our program operate until it made 300,000,000 guesses. Also note that our Probabilistic Password Cracker was still creating guesses when we stopped it. We choose 300,000,000 just as a benchmark number. The results are shown in Fig. 4.4.6.

These results match with the previous test on this data set, as seen in Fig. 4.4.1, in that given the same number of guesses our password cracker operating in terminal probability order cracks 68% more passwords than John the Ripper. As you can see in Fig. 4.4.6 though, the rate at which our method cracks passwords does slow down as more guesses are made. This is to be expected as it tries lower and lower probability password guesses.

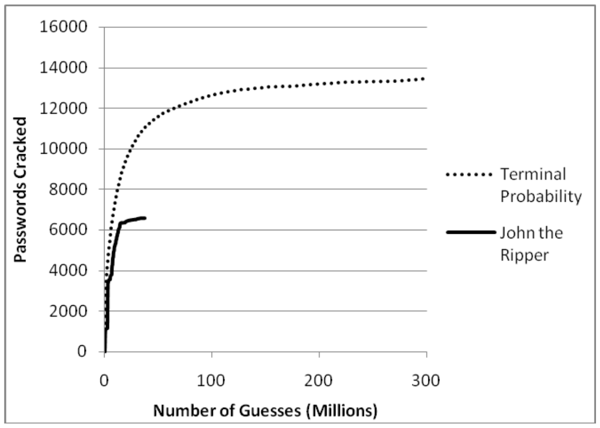

Fig. 4.4.6. Number of Passwords Cracked Over Time. Trained on the MySpace Training List. Tested on the MySpace Test List

We decided to run the test in Fig. 4.4.6 again, but this time have John the Ripper switch to brute-force after exhausting all of its word-mangling rules. We feel this would best simulate an actual password cracking session, (aka exhaust a dictionary attack and then resort to brute-force) using John the Ripper. Please note that John the Ripper uses Markov models in its brute-force attack to first try passwords phonetically similar to human generated words. It creates the conditional probability based not only on letters, but also on symbols and numbers as well. As a third experiment, we also ran a pure brute-force attack without using John the Ripper's rules first. The results of these tests are shown in Fig. 4.4.7.

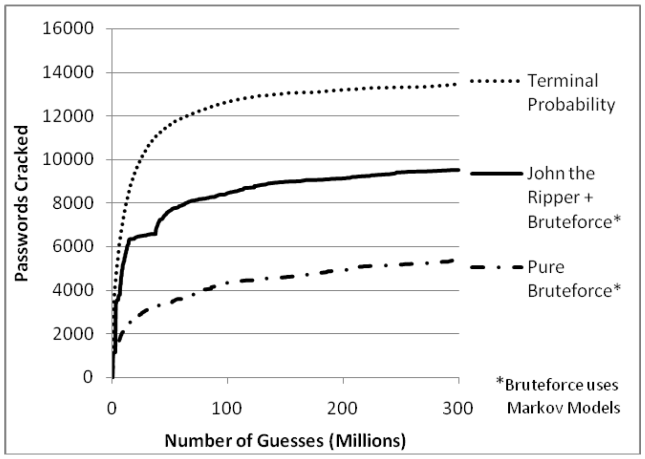

Figure 4.4.7. Number of Passwords Cracked Over Time. Trained on the MySpace Training List. Tested on the MySpace Test List

One thing we learned from this data is that it may be effective to pause our probabilistic method after around 100 million guesses and switch to a brute-force attack using a small keyspace for a limited time before resuming our probabilistic attack. This would allow us to quickly crack any short passwords our method may have missed. After a period of time though, brute-force becomes completely infeasible due to the length of the passwords and the size of the keyspace. We expect that even the low probability guesses generated by our cracker are better than a completely random guess which would result from a pure brute-force approach against a large keyspace. Therefore, the more passwords you can crack before having to solely rely upon brute-force attacks, the more advantageous it is. Because of this, the large number of rules, (possibly billions), that our method automatically generates is another major advantage of our approach.

4.5 SPACE COMPLEXITY RESULTS

In this section we focus on the space complexity related to generating our password guesses. We first discuss the space complexity of storing the grammar as this is what would be distributed to the end user once the password cracker has been trained.

Since the grammar is generated from the training set, the size of the grammar is dependent on the size of this set. To distribute this grammar we need to save the set of *S*-productions (grammar rules with the start symbol *S* on the left hand side) that give rise to the base structures and their associated probabilities. See Figure 3.2.1. Consider a training set of *j* passwords each of maximal length *k*. At worst each password could result in a unique base structure resulting in O (*j*) *S*-productions. Similarly the number of D_i-productions and S_i-productions depend on the number of unique digit strings and special strings, respectively, in the training set. This could result in a maximum of O(*jk*) unique productions. Finally, the number of L-productions (rewriting an alpha string using a dictionary word) depend on the input dictionary chosen. For a dictionary of size *m*, the maximum number of L-productions is simply O(*m*). In practice, we expect the grammars to be highly portable with many fewer production rules than the worst case. See Table 4.5.1.

Table 4.5.1
Size of the Stored Grammar

Training Set & Size	**# of Base Structures**	**Number of S_i-productions**	**Number of D_i-productions**
MySpace10k	820	79	2405
MySpace20k	1216	108	3377
MySpace (33,561)	1589	144	4410
Finnish (15,699)	736	49	1223

We next consider the space complexity of an actual password cracking session. Using the grammar, for each base structure, we generate pre-terminal structures, using the "next" function that are pushed and popped from the priority queue as described in Section 3.3. The space complexity of this algorithm is the maximum size of the priority queue. It should be clear that this is worst case O(*n*) where there are *n* possible pre-terminals generated. We do not expect that the worst case is actually sublinear. In practice, the maximum size of the priority queue has not been an issue in our experiments to date. Table 4.5.2 shows the total number of pre-terminals generated to create a specified number of password guesses. The space requirement is shown by the maximume size of the priority queue. Figure 4.5.3 shows the size of the priority queue as a function of the passwords generated when trained on the MySpace training sets.

Table 4.5.2
Space Cost using Dic-0294

Training Set	Total Pre-Terminals Generated	Maximum Size of Queue	Password Guesses (millons)
MySpace10k	28,457	1,274	50
MySpaceFull	14,661	1,688	50
Finnish	19,550	4,753	50
MySpace10k	174,165	4,642	500
MySpaceFull	109,453	3,691	500
Finnish	1,567,911	138,187	500
MySpace10k	470,949	9,946	1,000
MySpaceFull	193,963	6,682	1,000
Finnish	4,324,913	299,933	1,000

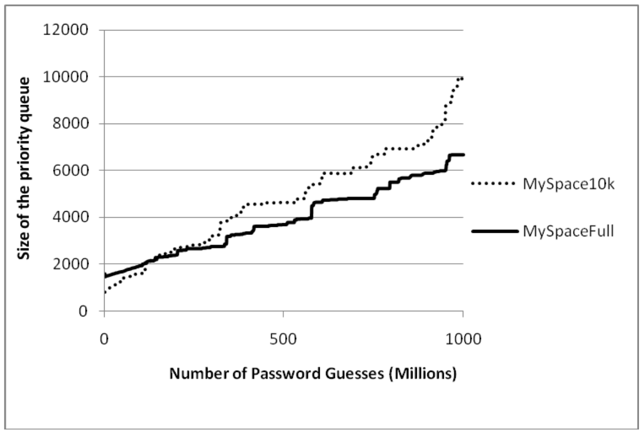

Figure 4.5.3 Size of the Priority Queue over Time, using Dic-0294 as the Input Dictionary

All tests were run using terminal probability order and using the dictionary Dic-0294. Note that in terminal probability order while the specific L-production is not expanded in the priority queue, its probability is taken into account when pushing and popping the pre-terminal structures. The input dictionary thus can cause differences in how the priority queue grows.

We finally consider the maximum number of pre-terminals and password guesses that could possibly be generated by our grammar. Consider as an example a base structure that takes the form $\mathbf{S_1L_8D_3}$. A pre-terminal value might take the form **$**$\mathbf{L_8}$**123**, and a final guess, (terminal value), might take the form **$password123**. To find the total number of possible pre-terminal values for this base structure, one simply needs to examine the total possible replacements for each string variable in the base structure. Using this example, and assuming there are 10 S_1-production rules and 50 D_2-production rules, then the total number of pre-terminals that may be generated by $\mathbf{S_1L_8D_3}$ is 500.

To find the total number of password guesses we simply expand this calculation by factoring in the number of dictionary words that can replace the alpha string. In the above example, if we assume there are 1,000 dictionary words of length 8, then the total number of guesses would be 500,000. See Table 4.5.4 for the total search space generated by each given training set and input dictionary.

Table 4.5.4
Total Search Space

Training Set	Input Dictionary	Pre-Terminals (millions)	Password Guesses (trillions)
MySpaceFull	dic-0294	34,794,330	>100,000,000
MySpaceFull	English-Wiki	34,794,330	>100,000,000
MySpaceFull	Common_Passwords	34,785,870	36,000
Finnish	dic-0294	578	>100,000,000
Finnish	English-Wiki	578	10,359,023
Finnish	Common_Passwords	506	6

To explain the results of Table 4.5.4 further, note the number of pre-terminals generated can be dependent on the input dictionary, since if a L_i-production exists where no dictionary word matches it, (for example the dicionary does not contain any words of length 9), then the base structure containing the L_i-production is discarded for that password cracking run. Also, we found that the total number of pre-terminals were mostly driven by a few base structures that contained a large number of D_i and S_i-productions, for example $\mathbf{S_1D_3S_2D_3S_1D_4}$. Likewise the number of terminals, (final password guesses), was dominated by a few base structures that contained many L_i-productions such as:

$\mathbf{L_1S_1L_1S_1L_1S_1L_1S_1L_1S_1L_1S_1L_1S_1}$. This was made worse by the fact that in our code we did not remove duplicate dictionary words. For example we would have 52 $\mathbf{L_1}$ replacements from the input dictionary "dic-0294" even though we lowercased all input words before using them. This is because by not removing duplicates we had two instances of every single letter of length 1.

That being said, the advantage of our method is that these highly complex base structures will generally not be utilized until late in the password cracking session due to their corresponding low probabilities. Therefore, we would not expand them in our priority queue until all the more probable guesses have been generated first.

5. FUTURE RESEARCH

There are several areas that we feel are open for improvement in our approach with using probabilistic grammars for password cracking. As stated earlier in section 3.2, we are currently looking into different ways to do insertion of dictionary words into the final guess that take into account the size of the input dictionary. As can been seen in Figures 4.4.1 – 4.4.5, there was a definite advantage to using terminal probability order vs. pre-terminal probability order with our probabilistic password cracker. Currently we determine the probability of dictionary words of length n by assigning a probability of $1/k$ if there are k words of length n in the input dictionary. There are however many other approaches we could take. Currently the most promising approach seems to be the use of several input dictionaries with different associated probabilities. This way one might have a small highly probable dictionary, (aka common passwords), and a much larger dictionary based on words that are less common.

Another point where we have identified room for future improvement is modifying the base structures to more accurately portray how people actually create passwords. For example, we could add another category, 'U', to represent uppercase letters as currently our method only deals with lowercase letters. Also we could add another transformation to the base structure that would deal with letter replacement, such as "replace every 'a' in the dictionary word with an '@'." Since we are using a context-free grammar, this would be fairly straightforward. All we need to do is create a new production rule that deals with letter replacement. The harder part would be identifying those transformations during the training phase. We are currently looking into several ways to efficiently identify those transformations such as checking the edit distance between known passwords and a dictionary file.

It may also be useful to add probability smoothing or switch to a Bayesian method in the training stage. This would allow our generator to create password guesses of a structure or containing a terminal value that was not present in the training set. For example, currently if the number '23' does not appear in the training set, our method will never use it. Ideally we would like it to try this terminal value, but at a reduced probability compared to values found in the training set. The ultimate goal would be to allow our method to automatically switch between dictionary based attacks and targeted brute-force attacks based upon their relative probability of cracking a password. For example, it might try some word-mangling rules, then brute-force all words of length four, before returning back to trying additional word-mangling rules.

There also exists more research to be performed on verifying the performance of this method if it is trained and tested against password lists from different sources.

6. CONCLUSION

Our experiments show that using a probabilistic context free grammar to aid in the creation of word-mangling rules through training over known password sets is a promising approach. It also allows us to quickly create a ruleset to generate password guesses for use in cracking unknown passwords. When compared against the default ruleset used in John the Ripper, our method managed to outperform it by cracking 28% - 129% more passwords, given the same number of guesses, based on training and testing on the MySpace password set. Our method also did very well when trained on the Finnish training set and tested on the MySpace test set. Our approach is expected to be most effective when tailoring one's attack against different sources by training it on passwords of a relevant structure. For example, if it is known that the target password was generated to satisfy a strong password policy (such as requiring it to be 8 characters long and containing numbers and special characters) the algorithm could be trained only on passwords meeting those requirements. We have also shown that we can quickly and manageably generate password guesses in highest probability order which allows us to test a very high number of rulesets effectively.

We feel that our method might successfully help forensic investigators by doing better than existing techniques in many practical situations. Our work can also provide a more realistic picture of the real security (or lack of the same) provided by passwords. We expect that our approach can be an invaluable addition to the existing techniques in password cracking.

REFERENCES

[1] U. Manber. A simple scheme to make passwords based on one-way functions much harder to crack. Computers & Security Journal, Volume 15, Issue 2, 1996, Pages 171-176. Elsevier.

[2] J. Yan, A. Blackwell, R. Anderson, and A. Grant. Password Memorability and Security: Empirical Results. IEEE Security and Privacy Magazine, Volume 2, Number 5, pages 25-31, 2004.

[3] R. V. Yampolskiy. Analyzing User Password Selection Behavior for Reduction of Password Space. Proceedings of the IEEE International Carnahan Conferences on Security Technology, pp.109-115, 2006.

[4] M. Bishop and D. V. Klein. Improving system security via proactive password checking. Computers & Security Journal, Volume 14, Issue 3, 1995, Pages 233-249. Elsevier.

[5] G. Kedem and Y. Ishihara. Brute Force Attack on UNIX Passwords with SIMD Computer. Proceedings of the 3rd USENIX Windows NT Symposium, 1999.

[6] N. Mentens, L. Batina, B. Preneel, I. Verbauwhede. Time-Memory Trade-Off Attack on FPGA Platforms: UNIX Password Cracking. Proceedings of the International Workshop on Reconfigurable Computing: Architectures and Applications. Lecture Notes in Computer Science, Volume 3985, pages 323-334, Springer, 2006.

[7] M. Hellman. A cryptanalytic time-memory trade-off. IEEE Transactions on Information Theory, Volume 26, Issue 4, pages 401-406, 1980.

[8] P. Oechslin. Making a Faster Cryptanalytic Time-Memory Trade-Off. Proceedings of Advances in Cryptology (CRYPTO 2003), Lecture Notes in Computer Science, Volume 2729, pages 617-630, 2003. Springer.

[9] *A list of popular password cracking wordlists*, 2005, [Online Document] [cited 2008 Oct 07] Available HTTP http://www.outpost9.com/files/WordLists.html

[10] A. Narayanan and V. Shmatikov, *Fast Dictionary Attacks on Passwords Using Time-Space Tradeoff*, CCS'05, November 7–11, 2005, Alexandria, Virginia

[11] *John the Ripper password cracker*, [Online Document] [cited 2008 Oct 07] Available HTTP http://www.openwall.com

[12] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, 1979.

[13] L. R. Rabiner, *A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition*, Proceedings of the IEEE, Volume 77, No. 2, February 1989

[14] N. Chomsky. *Three models for the description of language. Information Theory*, IEEE Transactions on, 2(3):113–124, Sep 1956.

[15] Robert McMillan, *Phishing attack targets Myspace users*, 2006, [Online Document] [cited 2008 Oct 07] Available HTTP http://www.infoworld.com/infoworld/article/06/10/27/HNphishing myspace 1.html.

[16] Bulletin Board Announcement of the Finnish Password List, October 2007, [Online Document] [cited 2008 Oct 07] Available HTTP http://www.bat.org/news/view_post?postid=40546&page=1 &group.

APPENDIX 1

PSEUDO CODE FOR THE NEXT FUNCTION

```
//The probability calculation depends on if pre-terminal or terminal probability is used
//New nodes will be inserted into the queue with the probability of the pre-terminal structure acting as the priority value
For (all base structures) { //first populate the priority queue with the most probable values for each base structure
        working_value.structure = most probable pre-terminal value for the base structure
        working_value.pivot_value = 0
        working_value.num_strings = total number of L/S/D strings in the corresponding base structure
        working_value.probability = calculate_probability(working_value.structure)
        insert_into_priority_queue(priority_queue, working_value)   //higher probability == greater priority
}
working_value = Pop(priority_queue) //Now generate password guesses
while (working_value!=NULL) {
        Print out all guesses for the popped value by filling in all combinations of the appropriate alpha strings.
        For (i=working_value.pivot_value; i<working_value.num_strings;i++) {
            insert_value.structure=decrement(working_value.structure,i); //get next lower probability S or D structure at pivot value 'i'
            if (insert_value.structure!=NULL) {
                        insert_value.probability = calculate_probability(insert_value.structure);
                        insert_value.pivot_value = i
                        insert_value.num_strings = working_value.num_strings
                        insert_into_priority_queue(priority_queue,insert_value)
            }
        }
        working_value = Pop(priority_queue)
}
```