2010

Using Probabilistic Techniques to Aid in Password Cracking Attacks

Charles Matthew Weir

THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

USING PROBABILISTIC TECHNIQUES TO AID IN PASSWORD CRACKING ATTACKS

By

CHARLES MATTHEW WEIR

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Spring Semester, 2010

The members of the committee approve the dissertation of Charles Matthew Weir defended on March 31, 2010.

_____
Sudhir Aggarwal
Professor Directing Dissertation

_____
Washington Mio
University Representative

_____
Breno De Medeiros
Committee Member

_____
Mike Burmester
Committee Member

_____
Xiuwen Liu
Committee Member

The Graduate School has verified and approved the above-named committee members.

To my parents:
I'd have a longer dedication but they told me
to focus on the rest of this paper instead…
Thanks for all the good advice.

ACKNOWLEDGEMENTS

This research would not have been possible without the help, support and advice provided to me by Professor Aggarwal, and Professor De Medeiros. They are the reason why I use the word “we” more than “I”. The original idea to use probabilistic context free grammars to represent password creation was theirs, and I can’t count the number of hours they spent with me as we developed new techniques to move it from theory to reality. They challenged me to be my best and then provided the freedom for me to explore new avenues of research. In short, I was extremely fortunate to work with them, and I plan on continuing our collaborations in the future.

I’d also like to thank Bill Glodek, another graduate student, for his assistance. I had the pleasure of working with him on password cracking for a year before he graduated, and his research showed the validity of using probabilistic grammars to attack password creation strategies.

Finally, I would like to thank the National Institute of Justice for funding this research, along with the Florida Department of Law Enforcement and National White Collar Crime Center with providing support and direction. When explaining my research to other people, the most frequent question I’m asked is, “Why are you trying to improve password cracking techniques?” It gives me great pride to be able to respond that my work is helping to catch and prosecute criminals whose crimes range from child exploitation to insider trading. This focus on helping to solve real world crimes by assisting law enforcement bypass strong encryption has been one of the driving forces behind my research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

At its heart, a password cracking attack is just a guessing attack. An attacker makes guesses about a user's password until they guess correctly or they give up. While the defender may limit the number of guesses an attacker is allowed, a password's strength often depends on how hard it is for an attacker to model and reproduce the way a user created their password.

If humans were effective at practicing unique habits, or generating and remembering random values, cracking passwords would be a near impossible task. In reality, that isn't true. A vast majority of people still follow common patterns, from capitalizing the first letter of their password to putting numbers at the end. What is changing though are the protective techniques being employed that are independent of user behavior. Practices such as salting password hashes negate the ability to pre-compute attacks. Likewise, password hashes are becoming more computationally complex, raising the costs for each guess an attacker makes.

While before an attacker could rely on simple brute force methods and ad-hoc models, there is a growing demand for more effective ways to predict what a user's password will be. The need for this is especially strong in the law enforcement community, where tough encryption is encountered regularly. It is also important for the defender to be able to accurately model the security that user generated passwords provide.

This paper details several new ways that probability information can be applied to maximize the success of password cracking attacks. From evaluating the effectiveness of known probabilistic techniques such as Markov models, to designing novel techniques such as using probabilistic context free grammars to create password guesses, there are many different ways probability information can be incorporated into modeling user behavior. Furthermore, the techniques described in this paper have been developed using real life passwords and have been tested in actual controlled password cracking attacks. This focus on training and testing against large sets of real life passwords is fairly unique, and only possible due to the increasing availability of disclosed password lists. In addition to allowing the development of more effective attacks, knowledge of how people select passwords can then be applied to evaluating the effectiveness of password creation policies. For example, how much stronger is an eight character password compared to a seven character password? In short, a better understanding of how users create passwords can benefit both the attacker and the defender.

CHAPTER 1

INTRODUCTION

"But the files were encrypted, and the 39-year-old Wyllie refused to divulge the password. The inability of police to review the files - combined with the fact that a camera he used was unplugged when the raid was commenced - meant prosecutors lacked the hard evidence they needed to prove the man had secretly taped his flatmates." [1]

1.1 Overview of Password Cracking Research Detailed in this Paper

Human memorable passwords are an integral part of most computer security systems today. Passwords are essential to our online identity; we use them for everything from online banking to Facebook status updates. Therefore, accurately understanding the security that passwords provide us is highly important. In addition, as pointed out in the quote above, there are many times when people of trust possess legitimate reasons to crack a password. Much of the research in this paper has been supported by the National Institute of Justice due to the pressing need of law enforcement officials to be able to analyze evidence protected by strong encryption. In a large number of cases, ranging from prosecuting child exploitation to investigating insider trading, police possess encrypted files and the only way to identify their contents is to crack the password protecting them.

Over the course of this research it was startling to find the lack of public analysis into the effectiveness of password cracking strategies, despite the fact that some early works such as [61, 62, 79] demonstrated the potential impact and usefulness of such undertakings. Much of this almost certainly stems from the fact that until recently there were very few publicly available examples of real life passwords for researchers to work with. As we'll see in Chapters 1.5 and 1.6, this is starting to change though. As additional examples of how people create and use passwords become available, a more detailed scientific inquiry into the security that passwords provide can be attempted. That in turn allows new attack methodologies to be developed and evaluated.

When writing this dissertation, my goal was not only to document my research and the new techniques I have developed over the last three years, but also provide a reference to help understand this research in comparison to existing password cracking methods. Therefore, while Chapters 1.2 through 1.4 are mostly background information, the main points to take away from them is that due to stronger password hashes being used, the computational costs of a password cracking session is dramatically rising. At the same time, most existing password cracking tools possess only a rudimentary ability to model human behavior. In fact, of all the password cracking tools examined, only

two, John the Ripper and Access Data's PRTK, allow an attacker to customize their own word mangling rules. Chapter 1.5 covers the work I did collecting real user passwords along with the hacker subculture surrounding password cracking. Rounding out the first section is Chapter 1.6 which details the password datasets that will be referred to throughout this paper.

Chapter 2 delves into the effectiveness of different styles of brute force attacks, along with attempts to optimize brute force attacks using probabilistic methods. While most of the algorithms covered in this section are commonly known, to my knowledge this is the first public study of their effectiveness against large datasets of real life password. In addition, this section details one of the custom tools I wrote, middlechild, which applies word mangling rules to brute force attacks.

Chapter 3 covers traditional rule based dictionary style attacks. In it, I describe my work creating custom dictionaries from internet based sources such as Wikipedia, as well as dictionaries targeted specifically against passphrases as well as passwords. A majority of this chapter though deals with the development of a customized edit distance metric that can be used to reverse mangle known passwords. This has several very useful applications such as a password strength checker, finding the relative effectiveness of different input dictionaries, discovering new, (to the attacker), mangling rules, and generating mangling rules for use with traditional password crackers. This chapter also shows some of the limits of traditional rule based dictionary attacks, and why there is a need for more advanced methods such as those detailed in Chapter 5.

Chapter 4 builds upon Chapter 3 by describing a new algorithm I developed to create dictionary based rainbow tables which allow an attacker to pre-compute much of the work involved in a password cracking attack. While traditional brute force rainbow tables have been available for years, this method provides a way to extend the rainbow table algorithm to cover dictionary based attacks in a flexible manner while minimizing storage requirements.

Chapter 5 describes a completely novel take on performing dictionary based attacks using context free probabilistic grammars. While it may sound like an overstatement, I truly do believe that this will fundamentally change how people perform password cracking attacks. The basic idea is to move away from trying to design mangling rules that model user behavior, which is what traditional password crackers do. Instead we should focus on the probabilities associated with how users create their passwords. For example, we know people use words such as "password", "football", and "love", much more often than "zebra", and "xylophone". Likewise certain numbers such as "1234", are much more common than "8131". What the method I developed does is attempt to take this probability information and weaponize it to generate highly targeted password guesses.

Finally Chapter 6 deals with applying the knowledge learned from cracking passwords to helping the defender evaluate the strength of their password creation policies. More specifically it

covers some of my research into the effectiveness, (or lack thereof), of using entropy as a password strength measurement. While password entropy is basis for most password creation and use policies, this is one of the only existing investigations into the accuracy of using entropy as a security metric by cracking large sets of real passwords using existing password cracking techniques.

1.2 Overview of a Password Cracking Session

When talking about password cracking, it's important to keep in mind that there are two types of password cracking attacks: *online*, and *offline*. In an online password cracking attack, the attacker uses the targeted system like an oracle, submitting guesses to it in an attempt to gain access. The distinguishing feature of an online password cracking attack is the device or system the attacker is focusing on is operational, and security features set up by the defender, (such as those limiting the number of incorrect logins), are still active. A frequently cited example of an online attack is an attacker trying to log into someone else's website or computer account. Online attack can also include more esoteric examples such as defeating an electronic lock, guessing the pin number to a bank ATM, or gaining access to a locked cell-phone.

In an offline attack, the attacker has gained direct access to the password hashes or encrypted files. The reason why it's called an offline attack is that the system protecting these files has been bypassed, and thus the attacker is no longer limited by the defender's policies on the number of guesses they can make. Offline attacks are often seen when an attacker successfully breaks into a computer or a website and is trying to determine users' passwords. Cracking those passwords would then provide the attacker access to other sites and/or resources. Offline attacks are also employed in a forensics setting, such as described in Chapter 1.1 where law enforcement officers have obtained a computer hard-drive but must deal with encrypted files. An offline attack is then used to attempt to decrypt the files.

The main distinction between an online and offline cracking attack, (besides the tools used), is the number of guesses allowed to the attacker. This is important since if the attacker can only make a few guesses before getting locked out, even a very weak password can provide quite a bit of security. A good example of this is the PIN used with most bank ATM machines. While user passwords are typically limited to four digits, giving it a key-space of ten-thousand possible values, an attacker is generally limited to three of four incorrect guesses. In an offline attack on the other hand, the attacker is allowed as many guesses as they have time and computational power to make. This means that an offline cracking session comprising several billions guesses can be quite common depending on the password hash or encryption algorithm being attacked, and a four digit PIN would likely be cracked in

seconds. While the defender can attempt to make the guesses computationally expensive, they are always limited by the fact that this also affects legitimate users by increasing the time it takes to log in.

Due to the prevalence of offline password cracking attacks in computer forensics, most of the research described in this paper deals with offline attacks. As mentioned previously - in an offline password cracking attack the attacker has already gained access to the encrypted files or password hashes. It's important to note that in most cases the attacker is not attempting to break the encryption directly. Instead they are attacking the user password creation strategy by attempting to guess what that password was. Because of this focus on attacking the user, a password cracking attack can be viewed as a password guessing attack. This guessing attack can then be broken up into three different steps:

1. The attacker makes a guess of what the user's password might be. For example 'password123'
2. The attacker hashes that guess using whatever hashing algorithm is being attacked. In the case of file encryption, the hashing algorithm to convert a user's password to an encryption key is used.
3. The attacker then compares the hash of their password guess to the hash they are trying to crack. If the two hashes match, the password is considered broken. With file encryption, the attacker attempts to decrypt the file (or file header) with the key generated, and if the file is decrypted successfully, the password is considered cracked.

These three steps are repeated over and over until the attacker breaks the password; or runs out of time. A majority of this paper deals with the different ways to make guesses as discussed in Step #1. Step #2 will be covered in more detail in section 1.3. Finally, while step #3 may sound inconsequential, it does provide a significant stumbling block when attempting to crack several hundred thousand password hashes at the same time, or when a password salt is involved.

1.3 Password Hashes and Salts

A hash is a one way cryptographic function that takes data as input and returns a fixed length output. This is different from your typical encryption algorithms in the fact that there is no decrypt function for hashes. Ideally, any hash used to store passwords should be resistant to *pre-image* and *second pre-image* attacks. Pre-image resistance means given a hash, it should be infeasible for the attacker to use a weakness of the hashing algorithm to determine the text which created the hash. Second pre-image

resistance means given a hash, it should be infeasible for the attacker to create a collision involving that hash, such as creating a second password which would hash to the same value.

The above properties of a hashing function allow a computer to authenticate users via a password hash while at the same time limiting the ability of an attacker to log on even if they possess the password hash. When a user creates a password for the first time, the computer, (or program), hashes their password using whatever hashing algorithm it is using. The computer then stores the hash of the user's password, but not the actual password. Later when the user wants to log into the computer, they would type their password again. The computer would hash what they typed and then compare it to the hash it has on file. If those two hashes match, it then lets them log in. If the hashes are different, it alerts the user to try again.

Unfortunately, as described in Section 1.2, while it may be infeasible for an attacker to attack the hashing algorithm, they can still treat the hash like an oracle and perform an offline password cracking attack. If the attacker has a copy of the hash, they can make a guess of what the user's password might be, hash the guess, and then compare the guess's hash vs. the hash they are trying to crack. If the two hashes match, the attacker has cracked the password hash. If not, the attacker guesses again. Because of this, any hashing algorithm should always be viewed as a delaying tactic, and the security of it depends on the individual user's password selection.

As a side note, most file encryption protocols also use hashes, but in this case to render a user's password into a fixed length key. An attacker attacks file encryption in much the same way as password hashes, by making a guess, hashing the guess, but this time then using that hash to attempt to decrypt the file or the file header. If the file decrypts correctly, the attacker knows they were able to successfully crack the password. Throughout this paper while password hashes will be mentioned the most frequently, the same techniques also apply to cracking file encryption as well.

A *password salt* is a value added to a password hash to prevent hash lookup attacks. Hash lookup attacks, will be covered in much more detail in Chapter 4 "Dictionary Based Rainbow Tables", but the short description of them is that they are a way of pre-generating password hashes and then re-using the results in later password cracking attacks. A salt is simple a value added to a user's password sometime during the hashing process so that two users who pick the same password will have different password hashes. An example of that can be seen in Table 1.3.1, where the username is appended to a user's password before it is hashed.

Table 1.3.1 Demonstration of Password Salts

Salt Used	User	Password	Password Hash (MD5)
None	Bob	password1	7C6A180B36896A0A8C02787EEAFB0E4C
None	Alice	password1	7C6A180B36896A0A8C02787EEAFB0E4C
Username, (concatenated)	Bob	password1	484502C8A463B4EFD6DEC91D1996D403
Username, (concatenated)	Alice	password1	D9591F23A232750FCC80F7F134BD19C1

Password salts are important, since if the possible values for a password salt are large enough, it becomes infeasible for an attacker to pre-generate hashes using all the possible salts. In addition, unique passwords salts also make it harder for an attacker to target many different password hashes at the same time. Usually when attempting to crack lists of several thousand password hashes, the attacker only needs to hash their guess once, and then compare the hashed result against all of the targeted password hashes. If a unique salt is used though, the attacker has to hash each guess independently with the specific salt for each hash, before making a comparison. To give an example of this, consider an attacker targeting ten thousand password hashes. If a typical attack would take one hour to compete, and those password hashes did not possess a salt, it would take approximately one hour to attack all of them. If the passwords were hashed with a unique salt, the same attack would then take ten thousand hours or approximately 1.14 years to complete.

It's important to note that the password salt is not a secret; it is stored in plain-text on the server. The secret is the user's password itself. Because of this, the password salt can be thought of as an extension of the hashing algorithm. Also since the password salt is stored on the server, the user never needs to know that the password salt exists.

It's very hard to write a document about password cracking without talking about several specific password hashing algorithms. Much like a study of car engines will detail specific models and makes; a password cracking session is very dependent on the password hash being attacked. Depending on the computational complexity of the hash, the attacker may be able to make millions of guesses a second, or just one or two. Likewise, if a password hash is salted or not can have a huge impact on if hash lookup tables can be used, and whether auditing large lists of password hashes is feasible or not. The password hash also determines which password cracking tools may be used, as different tools will only support attacking certain hash types. To that end, below is a quick description of some of the most common password hashing algorithms encountered.

1.3.1 The UNIX Crypt Family of Hashes

The Crypt family is a suite of UNIX tools commonly used to hash and/or encrypt files [13]. The original Crypt(1) algorithm was used to encrypt files, but is very rarely employed due to multiple weaknesses inherent to it. In fact, tools to crack Crypt(1) have been around since 1984 [14]. The specific password hashing algorithms currently in use of the Crypt family are labeled Crypt(3). The original version of this used a modified form of DES to create the password hash, and salted that hash with a twelve bit number. It proved fairly secure, but has fallen out of use due to the speed at which guesses may be made. Some system administrators took their own initiative to run the Crypt(3) algorithm multiple times on each password to increase the time requirement and thus make it stronger [13]. This had the added advantage of making their implementation of the Crypt(3) incompatible with the standard and thus many password cracking programs would not be able to attack them without modification.

Additional versions of the Crypt hashing algorithm have become popular, and use multiple rounds of hashing algorithms such as MD5, SHA1, SHA2 or BlowFish, as their base. The specific algorithm being used, (along with the number of rounds it is run), can be identified by the first values of the password hash. For example, passwords hashed by a thousand rounds of MD5 may be identified by their hash starting with a '1'. Likewise, hashes using the BlowFish algorithm start with a '2', and hashes using the SHA1, and SHA2 start with a '5', or a '6'. It is widely considered that modern versions of the Crypt(3) password hash are extremely resistant to attacks [15], but their security still depends on user password selection.

1.3.2 The LANMAN Hash

The Microsoft Windows LANMAN, (also known as LM), password hash is extremely prevalent, and unfortunately tremendously weak [16]. As the main password hash used when DOS was developed, it was preserved in all future releases up to Window Vista where the hash is no longer enabled by default. While the ability to disable the use of LANMAN hashes has existed since Windows NT, it required a manual change to the registry file which few users performed. Thus, the password hashes on most Windows computers and servers has been trivial to crack for quite some time.

There are several reasons why the LANMAN hash is considered broken. Much like the original Crypt(3), it is based off of a modified version of the DES encryption scheme. Therefore, an attacker can make guesses very quickly. Also no password salt was used making it vulnerable to pre-computation attacks. Furthermore it severely downgraded the strength of any password the user selected due to several poor design choices. First it only allowed passwords to be a maximum of fourteen characters

long. Worse, it then split the password hash up into two different hashes, the first one hashing the first seven characters of the password, and the second hashing the second seven characters. This meant that even a fourteen character long password was only as hard to crack as a seven character long password. Finally, it would convert all characters in the password to uppercase before hashing them. This means even if the user selected both lower and uppercase characters for their password, as far as the computer was concerned everything was uppercase. All of these weaknesses combined made it very easy to brute-force the entire key-space, meaning most LANMAN hashes can be cracked in less than thirty minutes even with the attacker using a desktop computer.

1.3.3 The NTLM Hash

The NTLM password hash is Microsoft's replacement for their LANMAN Hash. It was released with their Windows NT operating system, (hence the name, NT LANMAN), and has become the only password hash supported by default for local login since Windows Vista was released. While much stronger than LANMAN thanks to the fact that it avoids many of the design mistakes, NTLM uses one round of MD4 as the basis for its hashing algorithm which allows attackers to generate guesses fairly rapidly. Also, the NTLM hashing algorithm still does not make use of a password salt. On the plus side, the maximum password length was increased to 127 characters [17]. Also, it supports uppercase, lowercase and special characters along with numbers. This allows individual users to create extremely difficult to crack passwords. That being said, while NTLM is stronger than LANMAN, it still leaves a lot to be desired.

1.3.4 The WPA and WPA2 Hashes

WPA and WPA2 stand for 'Wifi-Protected-Access' and are encryption protocols used for wireless connectivity. While not a hash themselves, these protocols do exchange password hashes between the client and the wireless access point when a client connects. Due to the hash exchange mechanism, these hashes are vulnerable to interception by a malicious attacker, and are then subject to offline password cracking attacks. A redeeming factor though is that the hashes themselves are extremely computationally expensive to calculate, (4096 rounds of SHA1), which strengthens them against brute force. In addition, they enforce a strict requirement of the user's password being at least eight characters long, further limiting the effectiveness of brute force attacks against it. The hashes also are salted, but unfortunately they are salted with the SSID value of the access point. Since many people

choose to use the same SSID value, hash lookup tables have been created to attack the more popularly chosen access point names [20].

1.3.5 Website Password Hashes

Rather than use a custom designed password hash, many online websites instead use one of the commonly available general purpose hashing algorithms such as a single round of MD5 or SHA1 [3]. While not inherently flawed, these hashing algorithms were designed to be fast to compute. This is because one of the primary purposes they were intended for was to create message verification values and checksums. This lack of computational complexity in standard hashing functions is a serious weakness when used for storing passwords. Occasionally these password hashes are salted which greatly adds to their strength. Ultimately it is up to the site administrator to determine how they actually are used.

That being said, there are several other standard password storage techniques used in popular web forum software that use multiple rounds of common hashing algorithms. Named after the forum software they are included in, hashes such as vBulletin and PhpBB3 are becoming much more prevalent. vBulletin hashes are salted with a random value, and include two rounds of MD5. PhpBB3 hashes instead use a variable number of rounds of MD5, (often exceeding several thousand), and are also salted with a random value [18]. This standardization of online hashing algorithms is a positive move, as site administrators no longer have to develop their own password hashing techniques. While the uniformity helps the attacker as well, by allowing for the development of more specialized attack tools, it at least ensures that on average, passwords are stored in a more secure manner then if designing the password storage methods was left up to each site administrator.

1.4 Existing Password Cracking Tools

As can be imagined, there are many existing tools available for password cracking. For the most part though, the main difference between these tools is not the techniques they employ but they password types they support. The question, “Does it work against the hash I’m attacking,” generally drives which tool is selected for a password cracking session. Other considerations such as distributed support, the speed at which the tools can make guesses and the hardware the tools works on are also taken into account. For example many password crackers can perform their hash calculations on the CPU, (Core Processor Unit), GPU, (Graphical Processor Unit), or FPGAs (Field Programmable Gate Arrays). The above does not only cover offline password cracking tools. Programs such as THC Hydra [40], and

NCrack [41], are specifically tailored to attack network services and online websites. These programs are optimized to perform online password cracking attacks with network scanning ability and other features built into them. Because they perform online attacks, these tools also are generally run using very small input dictionaries due to the fact that they are often only allowed a few guesses against each online target.

Since the focus of this paper is on offline password cracking attacks, it's worthwhile to spend a little bit of time covering some of the more popular offline password cracking tools since they will be referred to throughout this paper. Please note that this isn't an exhaustive survey, but the tools covered provide a useful general guideline of the current password cracking software available.

1.4.1 John the Ripper

John the Ripper [23] is one of the oldest still maintained password cracking programs available. Originally based off the Crack program, it's main target was Unix based Crypt(3) hashes. It has since been expanded to handle most computer log-in and web-based password hashes. It does not include support for cracking any type of file encryption however. A screenshot of the program can be seen if Fig. 1.4.1.1.

```
Ryoki:run cweir$ ./john
John the Ripper password cracker, version 1.7.3.1-all-6
Copyright (c) 1996-2008 by Solar Designer and others
Homepage: http://www.openwall.com/john/

Usage: john [OPTIONS] [PASSWORD-FILES]
--single                  "single crack" mode
--wordlist=FILE --stdin   wordlist mode, read words from FILE or stdin
--rules                   enable word mangling rules for wordlist mode
--incremental[=MODE]      "incremental" mode [using section MODE]
--markov[=LEVEL[:START:END[:MAXLEN]]] "Markov" mode (see documentation)
--external=MODE           external mode or word filter
--stdout[=LENGTH]         just output candidate passwords [cut at LENGTH]
--restore[=NAME]          restore an interrupted session [called NAME]
--session=NAME            give a new session the NAME
--status[=NAME]           print status of a session [called NAME]
--make-charset=FILE       make a charset, FILE will be overwritten
--show                    show cracked passwords
--test                    perform a benchmark
--users=[-]LOGIN|UID[,..] [do not] load this (these) user(s) only
--groups=[-]GID[,..]      load users [not] of this (these) group(s) only
--shells=[-]SHELL[,..]    load users with[out] this (these) shell(s) only
--salts=[-]COUNT          load salts with[out] at least COUNT passwords only
--format=NAME             force hash type NAME: DES/BSDI/MD5/BF/AFS/LM/NT/XSHA/PO/raw-MD5/IPB2/raw-sha1/md5a/hmac-md5/KRB5/bfegg/nsldap/ssha/openssha/
oracle/MYSQL/mysql-sha1/mscash/lotus5/DOMINOSEC/NETLM/NETNTLM/NETLMv2/NETHALFLM/mssql/mssql05/epi/phps/mysql-fast/pix-md5/sapG/sapB/md5ns/HDAA
--save-memory=LEVEL       enable memory saving, at LEVEL 1..3
Ryoki:run cweir$
```

Figure 1.4.1.1 Screenshot of John the Ripper Password Cracker

John the Ripper is the password cracker covered the most throughout this paper. There are many reasons for this, but the main one is that it is an open source project. The advantage of this is that it was very easy to look at the code to see exactly what John the Ripper was doing. In addition, John the Ripper supports the ability to pipe guesses into it, which means it is possible to write a custom algorithm to generate password guesses, and then use John the Ripper as the backend cracker. Also it

includes the ability to export guesses generated from the built in algorithms to other programs, which made it convenient to map the effectiveness of a password cracking session by keeping track of exactly how many guesses were required to crack each password.

Beyond that, John the Ripper supports many advanced password guess generation techniques, and due to community involvement represents the current state of the password cracking field. It also has been used as the basis for previous password cracking studies, [28, 29, 43,], which allows new results to be compared against other analysis.

1.4.2 Cain & Able

Cain & Able [24] is a very widespread password cracking program with an extremely large fan-base. The reason for its popularity is that it runs on windows computers, is free, has an easy to use graphical interface, and more importantly incorporates many other tools directly into it. One popular feature included in it is a network sniffer that automatically grabs passwords and password hashes that it sees. If it captures a password hash, it can then automatically perform a password cracking attack against it. This feature is made even more powerful due to the fact that Cain & Able also includes the ability to perform an ARP poisoning attack. In this attack, the attacker sends false ARP reply packets to computers on the local network so they forward all their network traffic to the attacker's computer. Even more interesting, it also supports several man in the middle attacks against https secured traffic. Putting all of this together, Cain & Able is not only a password cracking program, but is also highly effective at collecting passwords and password hashes from targets on the local network. A screenshot of the program can be seen in Figure 1.4.1.2:

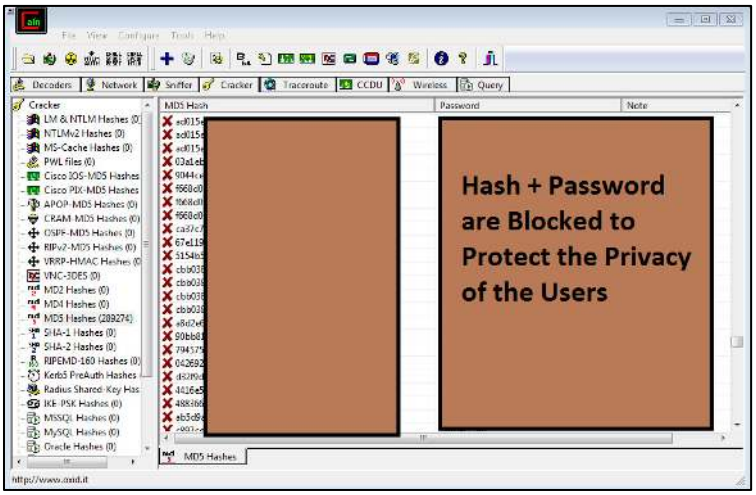


Figure 1.4.2.1: Screenshot of Cain & Able Password Cracker

Despite this added functionality, Cain & Able's actual password cracker is fairly limited. While it has built in support for creating Rainbow Tables, (which will be discussed more in Chapter 4), and has the ability to submit password hashes to online hash lookup databases, it has a very limited word mangling rule selection. The only rules it supports are case mangling, swapping letters with digits, and appending digits to the end of a password guess. This means it lacks the ability to attack any sort of strong password. Furthermore, its brute force methods only support letter frequency analysis attacks which is also very limiting. In short, while Cain & Able is a very well designed program, it really is only useful for cracking weak passwords.

1.4.3 L0phtcrack

L0phtcrack [44] gained a lot of notoriety back in the year 2000 when it was one of the first password cracking programs that could attack Windows LM hashes. Since then it was purchased by Symantec, shut down, re-purchased by the original creators, and re-released. L0phtcrack is mostly marketed to professional penetration testers who are performing security audits on company networks. Therefore it has a very well designed GUI, and the ability to create executive reports. A screenshot of it can be seen in Fig. 1.4.3.1:

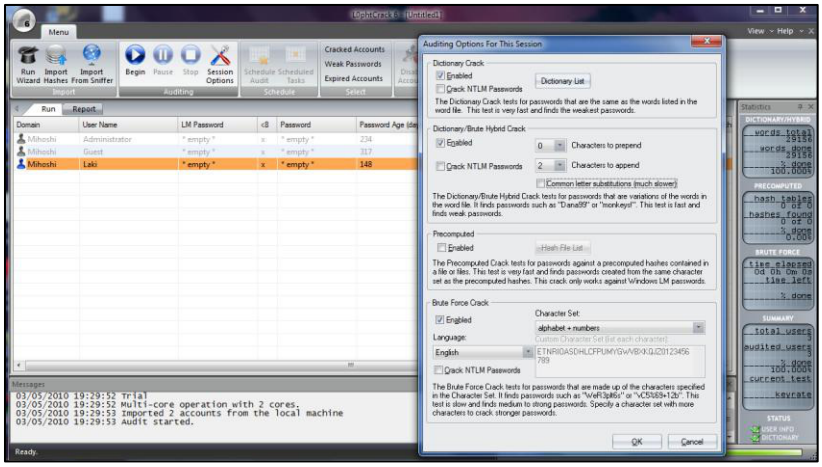


Figure 1.4.3.1: Screenshot of L0phtcrack Password Cracker

While the built in cracking rules are more sophisticated than those found in Cain & Able, L0phtcrack still lacks the ability to define custom word mangling rules or perform brute force attacks more sophisticated than letter frequency analysis. As mentioned before, L0phtcrack puts its emphasis on performing standardized attacks as part of a risk assessment. It's not designed to crack strong passwords.

1.4.4 Elcomsoft Distributed Password Recovery

Elcomsoft was one of the first companies to produce a password cracking program that not only could be distributed across multiple computers, but also takes advantage of a computer's graphical processor unit, (GPU), to hash password guesses as well. Also, unlike the password cracking programs previously mentioned, Elcomsoft's main password cracking program EPDR [22] is designed to crack file encryption along with windows log-in passwords. A screenshot of their program can be seen in Fig. 1.4.4.1:

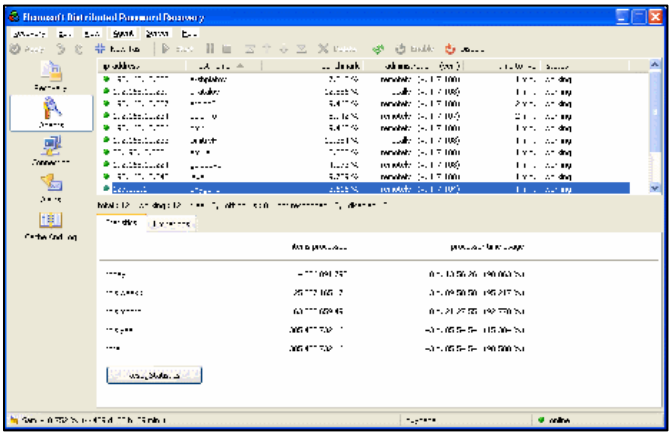


Figure 1.4.4.1: Screenshot of EPDR Password Cracker
Taken from the Elcomsoft Website [22]

Despite, (or perhaps because of), the power EPDR gives the attacker to make a large number of guesses extremely fast, it suffers the same problem that many other password cracking programs do, in that it not allow the attacker to specify their own custom word mangling rules to use in a dictionary based attack. That being said, they do support a respectable range of word mangling rules such as case mangling, letter replacement, and appending/pre-pending values. For brute force attacks, it only supports letter frequency analysis and not the more advanced techniques such as Markov modeling or targeted brute force attacks.

1.4.5 AccessData Password Recovery Toolkit

Like Elcomsoft, AccessData also produces a distributed password cracking program, PRTK [21], for cracking file encryption and password hashes. Unlike Elcomsoft, their program is designed to work with field programmable gate arrays (FPGAs), instead of GPUs to speed up a password cracking attack. A screenshot of the PRTK program can be seen in Fig. 1.4.5.1:

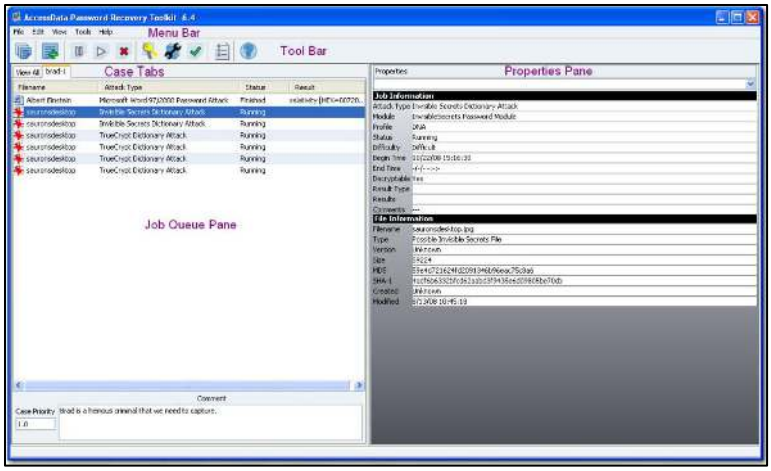


Figure 1.4.5.1: Screenshot of PRTK Password Cracker
Taken from the PRTK User's Manual [21]

What stands out about PRTK is that it has a fully customizable method for creating dictionary and brute force style attacks. For the brute force attacks, it supports both Markov modeling and targeted brute force. What's more, it allows brute force and dictionary attacks to be interlaced, where it switches between the two depending on the attacker's methodology. The only downside is that PRTK does not allow guesses generated by outside programs to be used as input like John the Ripper does. For the purposes of the research in this paper, a simulator program was created using the default ruleset available online in the PRTK User's manual [21], so that the normal attack methodology used by PRTK could be compared with other methods.

1.5 Obtaining the Datasets

Originally when starting this research, one of my biggest worries was if we could obtain enough examples of how people create passwords in real life. At the time, the only dataset available to our research group contained roughly sixty seven thousand passwords [2]. While that may sound like a large number, it was from a single user base, (in this case, MySpace users), and we ran a high risk of overtraining our attacks. Since then, we have collected over thirty three million passwords and password hashes from multiple sites. These passwords have ranged from UNIX logins, to webmail passwords. In one case, I even found my own username/password publicly posted online [3]. That shouldn't be surprising, (even though it most certainly was), considering the large number of websites where I have created accounts. It would be foolish to expect that all of them were never broken into, and in all likelihood my password has probably been stolen several times over the years, with only this one instance being publicly posted online.

As you can imagine, this leads to many different types of privacy issues. To that end, for my research I separated the passwords from their usernames, and instead used a numeric designator or a hash of the username if I was comparing passwords between different lists. This was to protect the privacy of the users since I needed to avoid any correlation between an individual user and their personal password. I also never verified any of the passwords, (such as attempting to log into accounts with them). Nor did I use any publicly available exploits to gain access to any lists myself. The last two points should go without saying, but they bear explicit mention to prevent misunderstandings. To limit further dissemination of these password lists, throughout this paper they are cited by the announcements of their disclosure, and not with links to the actual files. Due to their use as a research tool as common datasets, these lists are available upon request to legitimate researchers who agree to follow similar ethical standards.

That being said, I feel it is important to document my efforts to obtain these lists if only because it shows an interesting picture of the underground hacking scene today. All of the passwords I have worked with are the result of a publicly disclosed hacking attack. This is possible due to the fact that many blackhat, (and by blackhat I mean malicious), hackers often love to brag, as can be seen in Fig. 1.5.1.

Figure 1.5.1: Example Website Defacement

While site defacements as shown above are popular, another option for an attacker is to publicly post data about the website's users so the attacker can A) Prove they had full access to the site, and B) Attempt to cause harm to the website by causing the users to lose trust in the site's ability to protect their personal information. This information can be disclosed in a variety of ways. Examples include: posting the data to a public mailing list [2], posting the documents directly on one of the compromised

web-servers [4], and using public file sharing sites such as pastebin [5], torrents [6], and rapidshare [7]. Often these disclosures also include a write-up of the attack in addition to the password lists. One such example, posted by the ZF0 hacking group [4], can be seen in Figure 1.5.2:

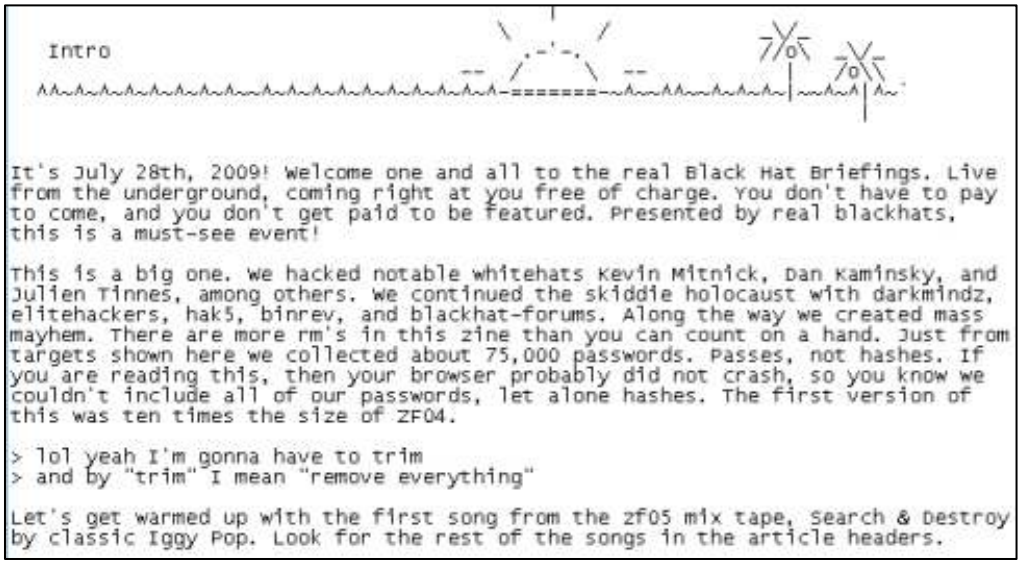


Figure 1.5.2: Example of a Hacker Write-Up

Such disclosures are becoming even more common as the venues for a malicious hacker to publicly brag about their exploits increases. One avenue for self promotion that has been gaining popularity is the BayWords blogging site, run by the creators of The Pirate Bay [6]. The blogging service has become widespread since BayWords promised in their mission statement that they will not delete blogs unless they break Swedish law, nor will they ever disclose the identity of the blog contributors [8]. Since most hacking attacks occur outside of Sweden, this has provided a public forum for hackers to release exploits and detail compromises against various websites without fear of having their site removed. Even a simple Google search using the query “inurl:baywords.com + sql” will return links to several hacking blogs, such as the one shown in Fig. 1.5.3. That site in particular was one of the places where the RockYou password list, (containing over 32 million passwords), was released [7]. The actual password files are no longer available on it though since while BayWords is a blogging service, they do not provide a file hosting. For that, blackhat hackers often store the password files on sites such as Pastebin, or Rapidshare, and simply share the links on their blog.

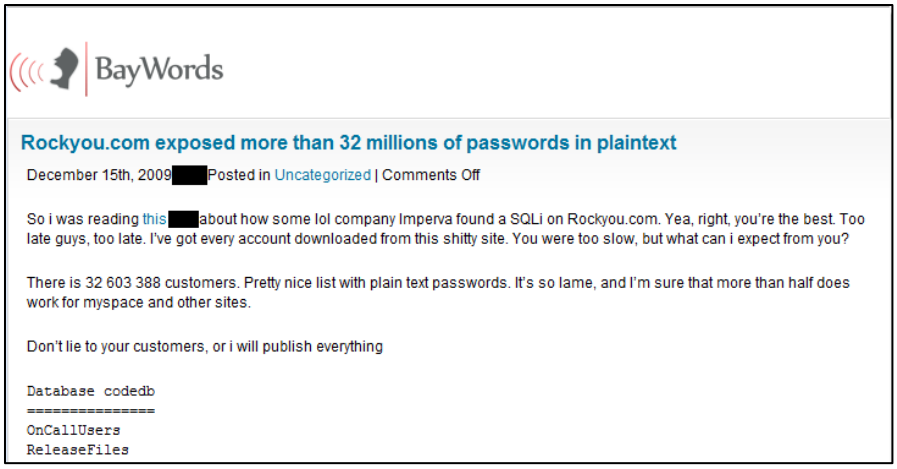


Figure 1.5.3: Example BayWords Hacking blog

Blogs of course are not the only place where these passwords are released and available for public collection and study. Blackhat forums such as [9] also provide an avenue for research, as hackers often brag about their exploits there. As an example of that, a screenshot taken from one of those forums can be seen in Fig. 1.5.4. In it you can see references to several posts containing webmail addresses and passwords/password hashes.

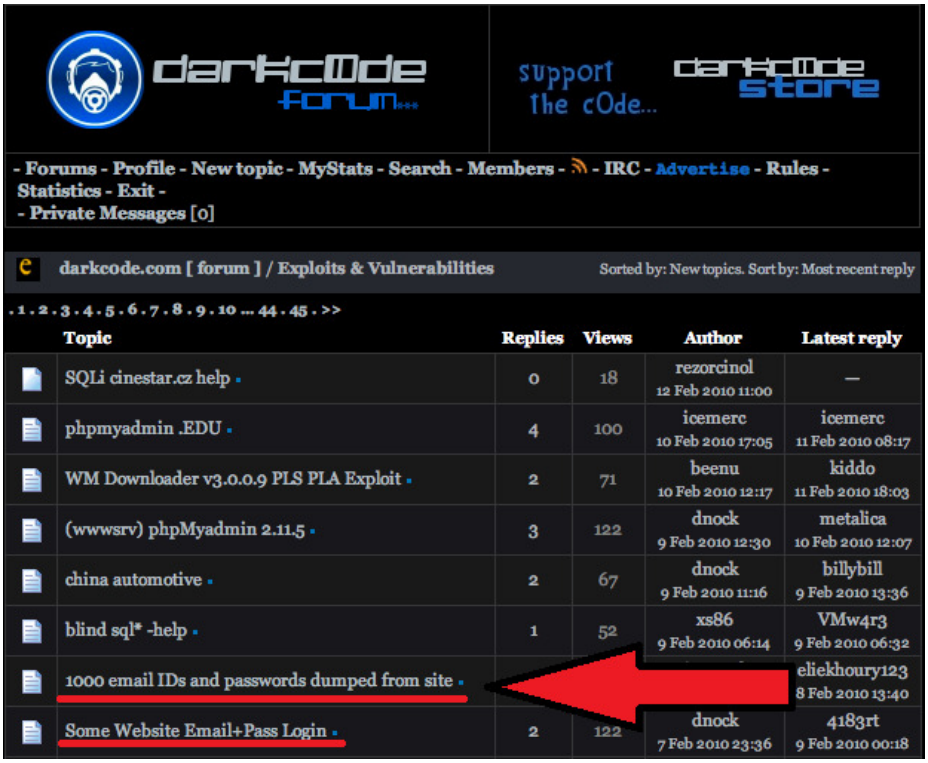


Figure 1.5.4: Example of a BlackHat Hacking Forum

There are many reasons why these password lists are released besides self-promotion by the original hackers. There are numerous websites that offer free password cracking services for common password hash types. Many of the blackhat hackers lack the expertise to crack the passwords themselves, and instead submit their collected password hashes to these sites and bulletin boards in an effort to have someone else crack the passwords for them. In fact, tools such as md5-utils [12] currently submits password hashes to thirty three different online password cracking websites in an attempt to break them. To show the volume of password hashes submitted to these sites, in Fig. 1.5.5 you can see a summary of the number of hashes submitted to one site in particular over a one week period as listed under “OpenCrack Daily”. In this particular week, the number of passwords submitted ranged from three thousand to a little over thirty thousand password hashes a day with a cracking success rate ranging from 11% to 45% of the passwords cracked. This particular site operates in collaborative fashion with various users competing to see who can crack the most passwords. In the right-hand column you can see the current “score” between these users. Since the un-cracked password hashes are posted publicly on the site, this gives us the opportunity to not only collect them, but also analyze other people’s password cracking techniques by analyzing the passwords they do manage to break.

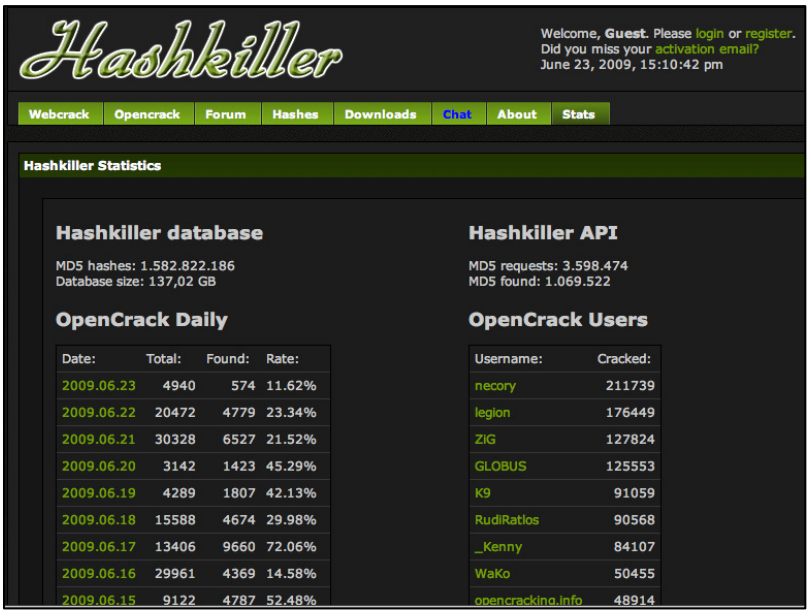


Figure 1.5.5: Example of an Online Password Cracking Site

It’s important to remember that the examples discussed in this chapter represent only a small fraction of the online hacking scene. I never participated in any blackhat hacking forums that required a

specific invite, or for me to contribute. Finally, it can safely be assumed that a vast majority of personal data collected by malicious hackers, (either via breaking into websites, malware keystroke loggers, or phishing attacks), is never disclosed publicly.

1.6 Description of Datasets Referenced

As mentioned in section 1.5, many of the passwords and password hashes collected over the course of this research were gathered from hacking forums and online password cracking websites. Unfortunately, while these lists are useful to gain insight into how people create passwords, there is often very little information surrounding the origin of these lists. This knowledge is important because people may choose different passwords based upon if they are visiting a banking site vs. an online gaming site. Also the passwords posted on many of the online cracking sites may only represent a subset of the passwords originally collected. For example a hacker may have only posted the passwords they could not crack, or in the case of bragging, the password they could crack. Furthermore, the posted list may only contain unique password hashes. This is a problem because people often choose the same password, such as '123456'.

That being said, there are a few specific password lists that have achieved widespread exposure. Because of that, they have been examined by multiple researchers and provide common datasets to analyze and test against. Therefore a vast majority of experiments detailed in this paper are run using these password lists. All of the lists are divided into training lists and test lists. This is to avoid training and testing against the same set. For example, if a user's password falls into the training list, it is not used as one of the target passwords in the test set. It should be noted though that it was not always possible to completely assure that the same user did not appear in both the training and test lists since several of the lists were not released with usernames associated with the passwords. In those cases it was assumed that each user only had one corresponding password and the sets were divided up accordingly. Also, for privacy issues, once the training and test lists were constructed all references to the original usernames were removed when performing a password cracking attack. This way when a password is broken, the corresponding plaintext password is not associated with a specific user. If a password list consisted of plaintext passwords, it could serve either as a training or a test list. If only the password hashes were retrieved though, then those hashes were classified as part of a test list since by definition the hashes had to be cracked before their corresponding plaintext values could be found. The following sections contain a quick description of several of the lists referenced throughout this paper.

1.6.1 The MySpace List

Originally published in October 2006 on the Full-Disclosure mailing list, the MySpace list was one of the first major password lists publicly disclosed [16]. The list was created by an attacker who had performed a phishing attack against MySpace users but failed to secure their collection server. Researchers proceeded to log into the malicious server and gathered the passwords from the hacker before the hacker could shut it down. The complete MySpace list contained 67042 plain text passwords. Unfortunately, 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 the test runs in this paper, there was no attempt to purge these fake passwords due to the difficulties in distinguishing between fake and real passwords.

Due to its size, and the fact this was one of the first password lists collected for my research, the MySpace list was divided into two parts, a training and a test list. This was done by giving each password a 50/50 percent probability of being assigned to a specific list. The end results were a training list containing 33,561 passwords, and a test list containing 33,481 passwords. Some of the basic statistics from the MySpace training list can be seen in Table 1.6.1.1:

Table 1.6.1.1: Password Characteristics of the MySpace Training List

Password Characteristics	**Results:**
Average password length	8.34 characters
Percentage that contain uppercase letters	6.92%
Percentage that contain special characters	9.49%
Percentage that contain digits	84.97%
Percentage that only contain lowercase letters	7.29%
Percentage that only contain lowercase letters and digits	84.08%
Percentage of passwords that contain lowercase, uppercase, digits, special characters and are greater than six characters long	0.26%

One thing to keep in mind is that at the time of the attack, MySpace had a password creation policy mandating that all passwords had to be at least six characters long and contain one non-lowercase letter. There were a number of passwords in this list that did not meet this requirement, but a

majority of those were most likely caused by users typing in the wrong password, or users who had originally created their passwords before the password creation policy was applied. That being said, what's really interesting is that when forced to create a password under this policy, most users choose to simply use one other character class, be it digits, uppercase characters, or special characters.

1.6.2 The PhpBB List

The PhpBB list originated from a hacker who broke into the phpbb.com website and posted all of the user information online along with a write-up of their attack [45]. Phpbb.com is the development site for the PhpBB bulletin board program which is used as the forum software for many other internet sites. The list itself contained 295,424 MD5 hashed passwords, along with over 83 thousand passwords hashed with the PhpBB3 hashing algorithm. The reason why two types of password hashes were used was because the site had changed their database software, but an individual's password hash was only converted to the stronger PhpBB3 hashing algorithm once they logged in after the upgrade. Since all of the passwords were hashed, they were all treated as a test set. A majority of the work done with these lists focused only on the MD5 hashed passwords due to the computational resources required to attack the PhpBB3 hashing algorithm. Therefore, when the PhpBB list is mentioned, unless explicitly stated otherwise, only the MD5 list is being referenced.

Over the course of this research, extensive work was done cracking these passwords to study the methods used in a long term password cracking session. A more complete coverage of the results of this study can be seen in the talk available online, "Cracking 400,000 Passwords or How to Explain to Your Roommate Why the Power Bill is So High" [18]. The short summary though is that initially it took approximately four hours to crack 38% of the passwords using one laptop computer. Adding a desktop computer to the mix, after one week 62% of the MD5 passwords had been cracked. After one month and one week 89% of the passwords had been cracked. The timeline is not as accurate past the one month period since cracking other lists took precedence, but the PhpBB list has been revisited several times as new improvements have been made to our probabilistic password cracking program, (described in more detail in Chapter 5). Also, since people often choose the same password for several different sites, as I collected more datasets this enabled me to crack more passwords since I saw the same users across several different datasets. At the time of this writing, I have cracked over 97% of the total MD5 hashed passwords in the PhpBB list. Some of the statistics learned from the cracked 97% of the passwords can be seen in Table 1.6.2.1.

Table 1.6.2.1: Password Characteristics of the Cracked PhpBB List

Password Characteristics	Results:
Average password length	7.27 characters
Percentage that contain uppercase letters	7.21%
Percentage that contain special characters	1.44%
Percentage that contain digits	45.77%
Percentage that only contain lowercase letters	50.62%
Percentage that only contain lowercase letters and digits	84.08%
Percentage of passwords that contain lowercase, uppercase, digits, special characters and are greater than six characters long	0.11%

It's important to remember that the above results do not contain the 2.6% of the passwords I have not been able to crack yet. While the only thing we know about the un-cracked passwords is what they are not, on average it should be expected that they are fairly strong. What's interesting is that while phpbb.com did not have a password creation policy in place, about the same number of people used uppercase letters as in the MySpace list. Without that policy though, many fewer people choose to use special characters or even digits in their passwords, with over 50% of the cracked passwords containing only lowercase letters.

1.6.3 The Hotmail List

The Hotmail list was collected by a sophisticated phishing scheme against hotmail users, primarily Portuguese or Spanish speakers [5], and disclosed online via a posting on pastebin.com. What's interesting is that the passwords posted only cover a small range of users. Specifically the list only covers users who e-mail addresses fell into the range ara*** to bla***. It should go without saying then that this is only a small sample of the entire list that the attacker collected. While it's unknown why the list was originally posted online, one possible option is that the attacker was trying to sell the entire list, and used this snippet to prove they actually possessed the passwords. Looking at some of the other e-mail lists collected through this research, the range, (ara-bla), probably constitutes around 4% of all the total e-mail addresses. This means that there's a good chance this 10k sample was selected from a list containing roughly 250k password. One caveat to this analysis was that I used mostly English e-mail addresses to come up with that number, so my estimate may be may be wildly off.

Currently Hotmail has a password creation policy that requires passwords to be six characters long or longer. Once the passwords in the list that did not conform to hotmail's password creation policy were removed, the list contained 9744 unique username/password combinations. The list was then divided up into two parts, a training list and a test list, each containing 4872 passwords. Some of the statistics gathered from the Hotmail training list can be seen in Table 1.6.4.1:

Table 1.6.3.1: Password Characteristics of the Hotmail Training List

Password Characteristics	**Results:**
Average password length	8.75 characters
Percentage that contain uppercase letters	8.06%
Percentage that contain special characters	5.21%
Percentage that contain digits	51.99%
Percentage that only contain lowercase letters	43.04%
Percentage that only contain lowercase letters and digits	86.95%
Percentage of passwords that contain lowercase, uppercase, digits, special characters and are greater than six characters long	0.24%

1.6.4 The 78k Finnish List

This list was collected by a Finnish hacker through SQL injection attacks against numerous sites, most of which were hosted in Finland [3]. The entire list included over 78 thousand username and password combinations. Of those, 15,699 of the passwords were stored in plain text, which I designated as a training set. The rest of the passwords were hashed, but since multiple sites were compromised, this list actually contains several different password hash types. The focus of my cracking sessions was the 22,733 passwords which were stored as a simple MD5 password hash with no additional salting. Since no salt was used, these password hashes were also included when I was attempting to crack the PhpBB MD5 password hashes. During those cracking session, I managed to break 97.25% of the target hashes from the Finnish list. A record of some of the basic statistics of the passwords in the Finnish training list can be seen in Table 1.6.4.1, and the statistics for the cracked passwords in the Finnish MD5 test list can be seen in Table 1.6.4.2.

Table 1.6.4.1: Password Characteristics of the Finnish78k Training List

Password Characteristics	Results:
Average password length	7.63 characters
Percentage that contain uppercase letters	3.00%
Percentage that contain special characters	1.79%
Percentage that contain digits	44.82%
Percentage that contain non-ASCII values	2.49%
Percentage that only contain lowercase letters	51.64%
Percentage that only contain lowercase letters and digits	92.82%
Percentage of passwords that contain lowercase, uppercase, digits, special characters and are greater than six characters long	0.15%

Table 1.6.4.2: Password Characteristics of the Finnish78k MD5 Cracked Passwords

Password Characteristics	Results:
Average password length	7.16 characters
Percentage that contain uppercase letters	7.42%
Percentage that contain special characters	1.46%
Percentage that contain digits	44.59%
Percentage that contain non-ASCII values	4.37%
Percentage that only contain lowercase letters	51.25
Percentage that only contain lowercase letters and digits	90.98
Percentage of passwords that contain lowercase, uppercase, digits, special characters and are greater than six characters long	0.11%

It's notable that even with 2.25% of the strongest passwords remaining un-cracked, the test list shares many characteristics with the training list. The only statistically significant difference was that uppercase characters were used much more frequently in the test list. Also, the category 'non-ASCII values' was added to the results to represent the number of Finnish characters, such as 'ä', and 'ö' that were encountered.

1.6.5 The RockYou List

The RockYou list [7] is by far the largest single set of publicly disclosed passwords to date. Containing over thirty two million plain-text passwords, none of the previous lists come anywhere close to it. The parent website which this list originated from, RockYou.com, makes applications for social networking sites such as Facebook, and MySpace. Unfortunately, RockYou had to store user passwords in plain text since there was no federated login system available for many of its partner sites. Due to a SQL injection vulnerability in one of their applications, RockYou's entire database was stolen by hackers and the user passwords were posted online.

One difficulty from a research perspective is that only the passwords were publicly released. This was due to the fact that while the passwords themselves were not worth very much to the hackers, when combined with the associated e-mail addresses they could be sold for quite a bit of money. With only the passwords to analyze, it made it very difficult to separate the list into smaller training and test lists since there was no way to ensure that a user's passwords only appeared in one list. To that end, for testing purposes, the list was randomized using the GNU shuf tool, and then broken up into thirty two smaller sub-lists containing one million passwords each. As of right now, only ten of these sub-lists have been assigned to either training or test sets. It was decided to hold back on analyzing the other twenty two lists to save them for future tests. To avoid confusion, the first five sub-lists, (RockYou1 - RockyYou5), have been classified as test lists, while the last five lists, (RockYou28 – RockYou32) are classified as training lists. Statistics taken from the RockYou32 training list can be seen in Table 1.6.5.1.

Table 1.6.5.1: Password Characteristics of the RockYou32 training list

Password Characteristics	**Results:**
Average password length	7.88 characters
Percentage that contain uppercase letters	5.95%
Percentage that contain special characters	3.48%
Percentage that contain digits	54.08%
Percentage that only contain lowercase letters	41.59%
Percentage that only contain lowercase letters and digits	90.76%
Percentage of passwords that contain lowercase, uppercase, digits, special characters and are greater than six characters long	0.14%

CHAPTER 2

APPLYING PROBABILITY TO BRUTE FORCE ATTACKS

2.1 Description of Brute Force Techniques

As shown in Chapter 1.4, brute force attacks are very common in most password cracking tools. It's important to note that for the purposes of this paper, a brute force attack includes any type of password cracking attack that does not make use of an input dictionary of human generated words. The reason brute force attacks are so popular is that most, if not all, input dictionaries only cover a fraction of the total words that are employed by users when creating their passwords. A more detailed description of the coverage normal input dictionaries provide against test sets of real passwords can be seen in Chapter 3, "Evaluating Dictionary Based Attacks". It's tempting to label all brute force attacks as "dumb", but in reality very complicated logic may be applied to them. Specifically, advanced probability and frequency information may be used to limit the key-space searched when attempting a brute force attack, while still achieving a high coverage of the target passwords. Before delving into the effectiveness of these techniques, it first is useful to discuss the different ways probability may be used to enhance a brute force attacks.

2.1.1 Pure Brute Force

Pure brute force is any brute force attack that does not use outside probability information that is not found inherently in the key-space being searched. While quite basic, it is currently the default brute force method found in many different password cracking programs today [22, 24, 25]. The number of guesses required to cover a given key-space using a pure-brute force attack is equal to $\sum_{k=0}^{n} x^k$ where **x** is the size of the character set, and **n** is maximum length of the passwords being targeted. This means to target longer passwords, an attacker often chooses a reduced character set to include in their brute force attack. Several example reduced characters sets from popular password cracking programs [22, 25] can be seen in Table 2.1.1.1.

Table 2.1.1.1: Example Character Sets for Pure Brute Force Attacks

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

A pure brute force attack makes guesses in the order determined by the alphabet given it. Thus an attack targeting six character passwords using the loweralpha character-set would make guesses starting with, “aaaaaa”, and ending with “zzzzzz”. Different password crackers though vary based on if they increment the leftmost or rightmost character first. For example some might make guesses in *right incrementing order*, “aaaaaa, aaaaab, aaaaac ...”, while others might make the guesses in *left incrementing order* “aaaaaa, baaaaa, caaaaaa”. This can have a drastic impact on how quickly passwords are cracked, as we will see in more detail when we talk about letter frequency analysis.

2.1.2 Letter Frequency Analysis

A letter frequency analysis attack attempts to use the frequency of characters appearing in a training set to increase the effectiveness of a brute force attack. Much like a pure brute force attack, a letter frequency analysis attack uses an input character set, and the cost of fully covering a key-space is still $\sum_{k=0}^{n} x^k$, where **x** is the size of the character set, and **n** is maximum length of passwords being targeted. The difference is that the order in which the character set is applied is modified by the frequency of the characters appearing in a training set. This is done by the attacker in an attempt to crack a majority of the target passwords while only searching a small fraction of the total key space.

It’s important to note that a letter frequency analysis attack is entirely dependent on the dataset it is trained on. Take for example the classic letter frequency analysis of the English language that was originally published in [27] and can be seen in Fig 2.1.2.1.

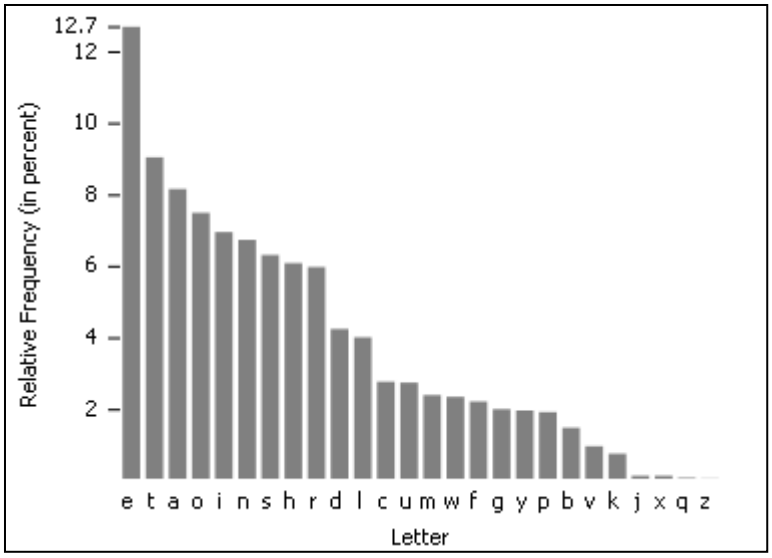


Figure 2.1.2.1: Example Letter Frequency Analysis of the English Language
Image published under the GNU Free Documentation License, Version 1.2

While the above figure is commonly cited as the standard letter distribution of the English language, it was actually generated by training on the novel Moby Dick along with several English poems. For whatever reason, Moby Dick is still one of the most popular training sets of the English language today, as can be seen in the intro to cryptography lecture slides from both Stanford [30] and Colombia University [31]. The question is, does Moby Dick represent the English language as a whole, and more importantly, does this represent the frequency at which certain letters are used in passwords?

The easiest way to answer that question is to perform some letter frequency analysis of disclosed password lists. You can see the letter frequency distribution gathered from real life password data-sets in Table 2.1.2.1. Notice that not only is the overall letter frequency distribution displayed, but the letter frequency distribution for the first and last letters of the password set is also included. That is because this information is useful to an attacker since the probability distribution of the first/last letter often is different from the overall probability distribution. A simple example of that can be seen in the fact that people tend to capitalize the first letter of their password, (if they use capital letters at all), and add digits to the end of their password. This means that the letter frequency distribution will differ if it is composed of all of the letters found in the passwords in the training set, or only the first or last letters. The letter frequency distribution used in a password cracking session then should be selected based on if the password cracking program employs a *right incrementing order*, or *left incrementing order* as described in Chapter 2.1.2.

Table 2.1.2.1: Letter Frequency Analysis of Various Password Lists

Training Set	Order	Letter Frequency Distribution (First 40 chars)
PhpBB.com	Overall	aeorisn1tl2md0cp3hbuk45g9687yfwjvzxqASER
PhpBB.com	First Letter	s1mpabctdrlfhgkjnw2ei0ov3q45796z8yuxSMPB
PhpBB.com	Last Letter	e1nsra326yt0d954o78lkgmihbpcxuwfzjvq!ESA
MySpace	Overall	ea1oirs1nt2cmd0hb3yugk9p6485f7wvjz!x.AES
MySpace	First Letter	sbmcpldatj1frhgkinwe2ovy304uB8S95z76MJDC
MySpace	Last Letter	1326795408!esary.ntdlohkmg*ucpibxjfzw?$v
RockYou22	Overall	ae1ionrls02tm3c98dy54h6b7kpgjvfwzAxEIORL
RockYou22	First Letter	sm1cba0pljdtrk2hgfniew39v45o8y76zMSBACJq
RockYou22	Last Letter	1ae326sn5794y08roltdihgmukzc!pxwbA.fEjS*
Hotmail	Overall	aeoi1r0ln2st9mc83765u4dbpghyvfkjAzEIOxRL
Hotmail	First Letter	a1mbc2sp0lterdjfgn3hi6k759vo48yAwMzBSCuq
Hotmail	Last Letter	aos01326e57849nrilydzmtuAhbO.gck*SxpfE@+
Example in Fig. 2.1.2.1	Overall	etaonrishdlfcmugypwbvkxjqz

There's a couple of interesting items that show up in the data presented in Table 2.1.2.1. First, the number '1' is extremely common as the last character of most English passwords, ranking either as the 1st, 2nd, or 3rd, most commonly used last character. In the Hotmail set though, which was composed primarily of Spanish and Portuguese speakers, the number '0' was more commonly used at the end of a password. Second, in the MySpace list, where a password creation rule mandating that passwords had to contain one non-lowercase character, the last letter was almost always a digit, or a '!". Finally, looking at the difference between the first letter, last letter and overall letter frequency distributions, it highlights the importance of selecting the right letter frequency distribution depending on how the password cracker increments its guesses.

2.1.3 Markov Models

Markov models have many uses, but in password cracking they are a way to represent the joint probability of different characters appearing together. This mimics their use in traditional language recognition algorithms [32], by using the joint probability of characters to generate human sounding words, (or in this case passwords), without the use of an input dictionary. The goal once again is to

perform a brute force style attack while minimizing the size of the key-space the attacker on average has to search through before they the successfully crack the target passwords. A *Zero Order Markov Model* is the equivalent of using letter frequency analysis as it has no memory. With a First Order Markov Model, the probability of each character after the first is given as $\mathbf{P(Q_t=S_i \mid Q_{t-1}=S_j)}$ where **i** and **j** represent the index of the character set used. A quick example of this is in the English language, if the letter ‘q’ appears, it is almost always followed by a ‘u’. Markov models can be extended beyond the first order to the second, third, a so on, as long as more state memory is used. Markov models beyond the second order are rarely seen in password cracking applications though due to concerns about overtraining on the input data-set and storage issues surrounding the resulting probability information.

While Markov models are extremely useful, the question becomes how should they be implemented in a password cracking program to produce effective password guesses? More to the point, while an attacker can use Markov models to very quickly evaluate the probability of a given guess, how do they produce those guesses in something resembling probability order? Currently there are two major algorithms being employed to do this, each with their own pluses and minuses.

The first method, made popular in the password cracker John the Ripper [23] is labeled *incremental mode*, Incremental mode actually models the probability of trigraphs appearing in the password guess. In this case a trigraph represents a three character long string, such as "abc". Another way to say this is that Incremental mode uses trigraphs to represent second order Markov probabilities, such as the probability of 'c' given 'ab'. The first two characters are handled as a special case, (since they don't have two characters before them), but after that John the Ripper select each successive character based on the previous two characters. Therefore, for the guess 'abcdef', it would create the trigraphs 'abc', 'bcd', 'cde', and 'def', and assign a final probability to the guess based on these four different trigraphs. In addition, the probability of a trigraph is also dependent on where it shows up in the password. In this way, the Incremental attack takes into account factors such as numbers often show up at the end of a password, and uppercase letters usually are at the front. The main downside with the algorithm is that it doesn't generate these guesses in true probability order, and instead uses an approximation. As we’ll see in later tests, this approximation results in it taking longer to crack passwords than a cracking session using the true probability order. The advantage of this optimization though is that given enough time, it will cover the entire targeted key-space, while still being able to generate a large number of guesses quickly.

The second method, first proposed in the Narayanan paper [19], and implemented in John the Ripper as *Markov Mode*, takes a different approach. Unlike incremental mode, the Markov mode does not use trigraphs. Instead it starts with the highest probability starting letter, (in the default stats file this is a 'c'), and then appends the most probable following character. It repeats this until it hits the

probability threshold specified, or it reaches 12 characters long, in which case it outputs a password guess. The attack then goes to the last character of the previous guess and changes it to the second most probable value, and outputs another guess. It continues to follow this algorithm as it cycles through printing out all the possible guesses that are above the probability threshold specified. There are many advantages to this approach. For example, it can attack passwords up to twelve characters long, and it is extremely fast. The downside of this approach is that it also does not create guesses in probability order. Instead it creates all the guesses within a certain probability threshold. Therefore, as we will see, if you set a low threshold, Markov mode starts to resemble a letter frequency analysis attack. Finally, because of this, Markov mode won't cover the entire key-space, (aka, it won't crack the password '3Nw!lswp'). The last point isn't as important as it may seem though, since in all likelihood due to time constraints, (aka the attacker only has several months to crack a password if that), the attacker generally won't have enough time to cover an entire non-trivial sized key-space even using incremental mode.

To demonstrate the difference between these two algorithms, both of the algorithms were run side by side against the Hotmail dataset with the results shown in Fig. 2.1.3.1. Each cracking session was allowed one billion guesses. Both the Markov mode and the Incremental mode used the default probability values included in John the Ripper. The limit for the Markov mode was set at '214', which would result in it generating slightly more than one billion guesses which have a higher probability than the threshold specified. For Incremental mode, the 'Allnum' character set was specified which resulted in it trying to brute force all lower-case characters plus numbers.

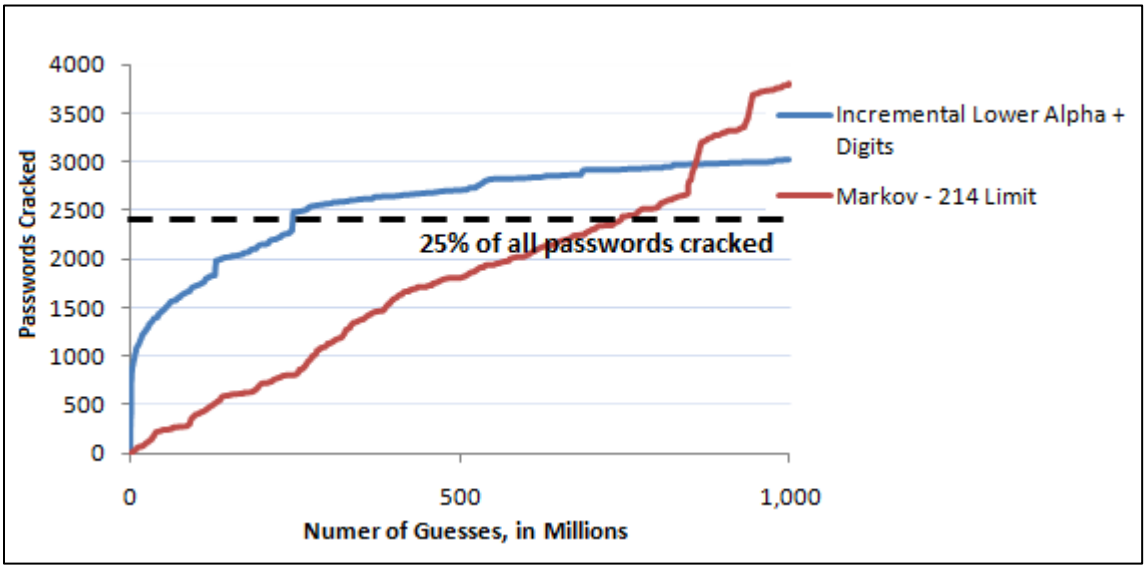


Figure 2.1.3.1: Incremental Mode and Markov Mode Run Against the Hotmail Dataset

As you can see, the Markov mode started out slow, but finished strong, cracking 786 more passwords than the Incremental attack. One of the reasons why it did so well near the end was that the Markov mode started trying password guesses that began with numbers, which was very common in the hotmail dataset. Also, since the probability of a guess starting with a number was so low according to the default rule-set, it only tried high probability strings following a number.

It's important to remember that due to the way it creates guesses - for the most part the Markov mode cracks passwords at a fairly fixed rate, while the Incremental mode is much more front-loaded. I'd like to caution you against extending the Markov mode's line though, as it has finished just about every password guess within the limit that was set. The next question then is: how does Markov mode perform when different limits are specified? For the next test, the same cracking session was re-run, but this time different limits were specified for the Markov mode attack. The results can be seen in Fig. 2.1.3.2:

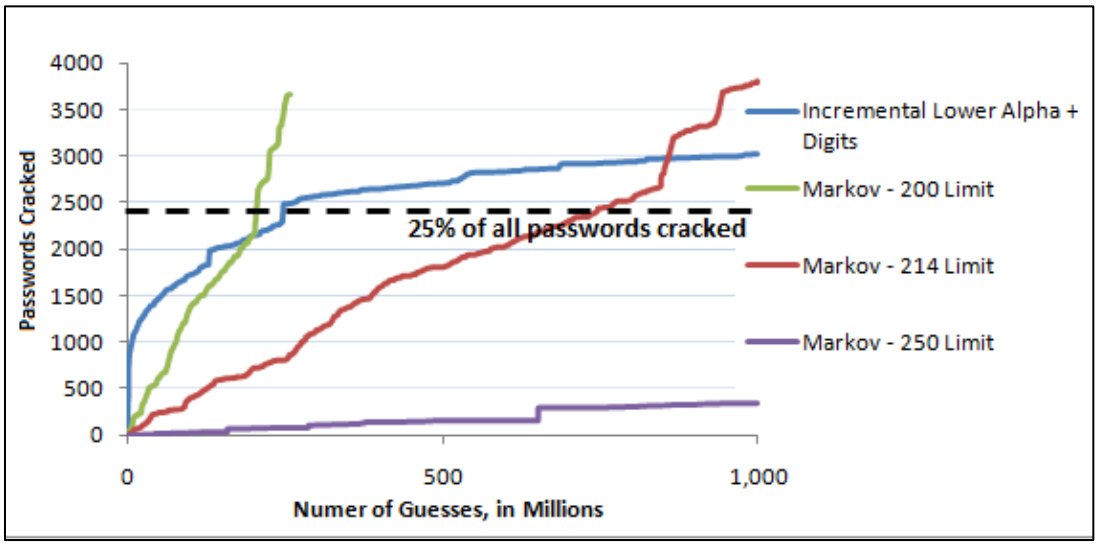


Figure 2.1.3.2: Markov Mode Run Against the Hotmail Dataset Using Different Limits

In this run, due to the way the algorithm is designed, Markov Mode cracks passwords at a fairly fixed rate, (and in fact operates almost exactly like a letter frequency analysis attack except for the fact that it has a limit). When using a limit of 200, it created slightly more than 250 million guesses and then finished. If we run the same attack using a limit of 214, it creates almost four times as many guesses, but it only cracks 148 more passwords. If we increase the limit to 250, it would generate roughly 46 billion guesses. That means if it was allowed to run to completion, the Markov attack would eventually catch up to Incremental mode, (and then exceed it). Over this short cracking session though, it

performed significantly worse, which is shown in the graph. What this demonstrates is that the attacker needs to spend the time to tailor their attacks using Markov mode to the resources they expect to spend cracking the target passwords. Otherwise, the Incremental mode algorithm is a superior option, since it automatically attempts to optimize itself to whatever number of guesses it is allowed. Another possible option for an attacker is to use multiple runs of the Markov mode algorithm, but add an upper probability limit where it rejects guesses that it made previously. In this way the attacker does not try to re-hash guesses they already made on a earlier run, while still cracking the higher probability passwords sooner. The downside of this approach is that it can significantly add to the overhead associated with making guesses as it generates the same guesses multiple times only to reject them. The finer grained an attacker makes these limits, the more times they would have to re-run this algorithm during a cracking session.

Now, the previous tests were run using the default probabilities included in John the Ripper. Unfortunately, those probabilities were created by training the two algorithms on different datasets, (the Markov mode was a user submitted patch). The Markov algorithm probabilities were trained primarily on French passwords, and the Incremental algorithm was mostly trained on English passwords. The question then is how they perform if they are trained on the same dataset. For the next test, I trained both the Incremental and Markov modes on the same set of disclosed passwords from the PhpBB list. In technical terms, I created a new .chr and new stats file for both attacks. The phpbb.com list isn't ideal for training to use against the Hotmail list, since the PhpBB list is primarily composed of English passwords, while Hotmail dataset is mostly composed of Spanish/Portuguese passwords. That doesn't matter for this test though, since what is important is both attacks are being trained on the same list, and they have to deal with the same limitations. For this test, I also ran the Markov attack twice. The first time I allowed it to attack passwords up to 12 characters long. The second time I limited it to only attack passwords up to 8 characters long, (and I assigned it a higher limit so it would generate roughly the same number of guesses). The reason for limiting the length was to better compare it against John the Ripper's incremental mode which only generates guesses up to 8 characters long. The results can be seen in Fig. 2.1.3.3.

Even when trained on the same datasets, Markov mode still cracks more passwords than John the Ripper's Incremental mode. Still, there were a couple of things that were surprising about the results. First of all, the phpbb.com list made a much better training list than using the default .chr and stat files. The Markov attack cracked 540 more passwords using the same number of guesses when it was trained on the PhpBB list. The Incremental attack also got off to a much faster start, though in the end it cracked 17 less passwords than the Incremental attack using the default Alnum character set. This isn't exactly a fair comparison though since this time it was trained to use the entire keyboard

instead of just lowercase letters and numbers. When compared to the default Incremental mode probabilities that contained the same character set, (aka all printable ASCII characters) it performs much better, cracking 333 more passwords overall. The second interesting point was that limiting the maximum password guess size didn't have much of an effect on the number of passwords cracked using the Markov mode. This shouldn't come as a surprise due to their low probability of Markov generated strings longer than eight characters. Therefore, very few strings larger than eight characters were generated that fell above the specified threshold.

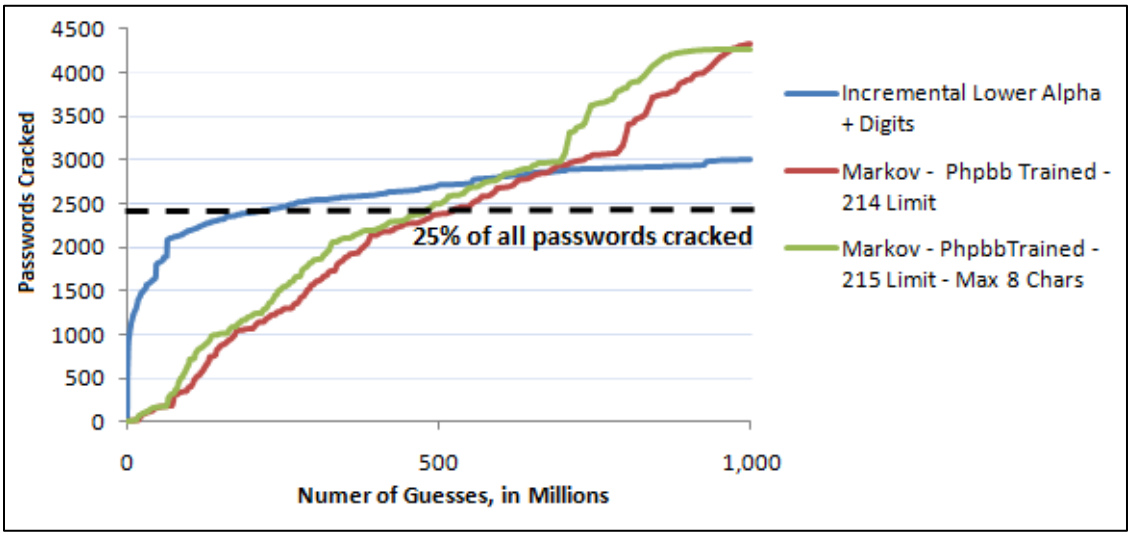


Figure 2.1.3.2: Markov and Incremental Mode Trained on the PhpBB Dataset and Attacking the Hotmail Dataset

Just as a reminder, both Markov mode and Incremental mode are Markov based attacks. They just are different algorithms that implement a Markov based attack given the time constraints that a password cracking program has when it comes to generating guesses.

2.1.4 Targeted Brute Force

A *Targeted Brute Force* attack can comprise letter frequency analysis and Markov models, but applies outside logic to these attacks. One example of a targeted brute force attack would be to perform a letter frequency analysis attack, but use a different character set for each character position. In this case, the first character could be determined by the first character distribution in the training set, the last character could be determined by the last character distribution in the training set, and the rest of the characters could be determined by the overall character distribution in the training set.

Another example of a targeted brute force attack is treat the input generated from a traditional brute force attack and input dictionary, and then apply traditional word mangling rules to them. One of the tools I created, labeled MiddleChild [33], does just that. Take for example the first few guesses output by John the Ripper's Incremental mode using the lower-alpha character set:

1. bara
2. sandy
3. shanda
4. sandall
5. starless
6. dog
7. bony
8. bool
9. boon
10. stark

While these guesses resemble human generated words, (and in some cases are actual dictionary words), they were generated by a brute force process. The problem is that guesses generated in this fashion are unlikely to crack any password created under a strong password policy that requires capitalization, numbers and special characters to be included. What MiddleChild does is take brute-force guesses as input and applies mangling rules such as capitalizing the first character, and then adding a special character and a number to the end. Examples of password guesses generated using the above words would be 'Bara$1', 'Sandy99#', '!Starless73', etc. These guesses now would be able to attack a strong password requirement, but they still are generated via brute force methods.

For a demonstration of the effectiveness of a targeted brute force attack please see the cracking session detailed in Fig. 2.1.4.1. In this test, two attacks are run against a subset of the Hotmail password list containing only passwords of length 8. Each attack was run for a total of one billion guesses. The first attack used John the Ripper operating in Incremental mode, which as mentioned in the previous section uses a very sophisticated Markov based approach. For this attack, the incremental mode was modified to only generate guesses of length 8 so it would not waste time making guesses that did not meet the password length requirements for this test. Also, incremental mode was run using the 'All' charset so it attempted to crack passwords containing letters, numbers and special characters. For the second attack, a targeted brute force approach was employed using the tool MiddleChild. Since there was no password creation policy in effect for the Hotmail dataset, (beyond the fact that the passwords had to be at least six characters long), several rules were used with MiddleChild to create its one billion guesses.

The following five rules were applied to generate the guesses using MiddleChild:

1. Eight Lowercase Letters: 250,000,000 Guesses
2. Eight Digits: 100,000,000 Guesses, (maximum number of guesses for the keyspace)
3. Seven Lowercase Letters, Followed by One Digit: 150,000,000 Guesses
4. Six Lowercase Letters, Followed by Two Digits: 250,000,000 Guesses
5. Four Lowercase Letters, Followed by Four Digits: 250,000,000 Guesses

All of the lowercase characters for the targeted brute force attack were generated using John the Ripper's Incremental mode operating in 'LowerAlpha' mode.

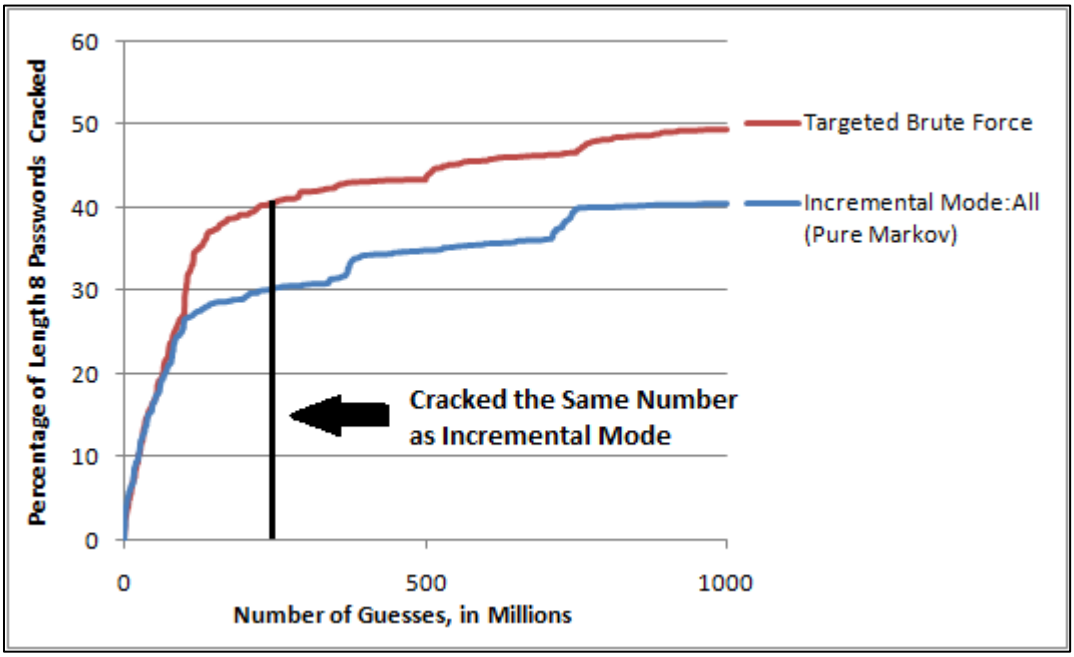


Figure 2.1.4.1: Targeted Brute Force Vs. Incremental Mode. Run on Length 8 Hotmail Passwords

The results show that the targeted brute force attack was able to crack the same number of passwords as Incremental mode using less than 242 million guesses. To put it another way the targeted brute force attack required around 25% of the guesses to crack the same number of passwords. It should be noted that further improvements can be made to the targeted approach since with the current version of MiddleChild, the order at which the digits are inserted is not optimized. For example, the numbers 0,1 and 9 are significantly more likely to occur than the number 5,6, and 7. Currently I am looking at including these optimizations in the probabilistic password cracker, UnLock, which will be described in more detail in Chapter 5.

It does bear mentioning a couple of other points about the test shown in Fig. 2.1.4.1. It is notable that even for length 8 passwords, almost 50% of them were cracked using a brute force attack with less than one billion guesses. Considering the key-space for passwords only containing letters and digits, (64 characters total), of length 8 is 218,340 billion possible passwords, this is a significant improvement as compared to a pure brute force attack. It also shows the feasibility of attacking 7, 8, 9, and even 10 character passwords using brute force attacks that incorporate probabilistic methods into them. In fact even longer passwords could be successfully cracked if the defender incorporated certain creation strategies, such as type the same eight character password twice, since the attacker can then model that using a tool such as MiddleChild.

2.2 Comparative Results

While it was briefly alluded to in the previous sections, it's worth analyzing how applying advanced probabilistic methods improves a password cracking session compared to a pure brute force or a letter frequency attack. For the first comparison I ran several brute force attacks against the hotmail password list. Each attack was allowed one billion guesses. The first attack used John the Ripper's incremental mode targeting lowercase letters and numbers, (a-z0-9). Once again, John the Ripper's Incremental mode uses 2nd order Markov probabilities to generate its guesses. For the second cracking session, a pure brute force attack was used. The third cracking session used letter frequency analysis to optimize its password guesses.

To simulate the brute force and letter frequency analysis attacks, I used the popular password guess generation tool crunch [47], since John the Ripper does not directly support pure brute force attacks. The command used to generate the pure brute force guesses was:

```
./crunch 6 8 abcdefghijklmnopqrstuvwxyz0123456789
```

When crunch is run using the above command, it generates brute force guesses between 6 and 8 characters long containing the characters a-z0-9, starting with aaaaaa and theoretically ending with 99999999, (in this run it never finished all of the six character long words). This is also the default character set that the popular password cracker Cain&Able [24] uses.

The letter frequency attack was based on analysis of the cracked phpbb.com password list. This was due to the fact that these passwords were viewed as a representative sample of passwords that users select when there is no existing password creation policy. For the specific probabilities, only the first character of the phpBB cracked passwords were considered. This is because crunch increments its

values from right to left; aka it tries 01, 02, 03, 04 etc. This means using the following command, crunch will initially crack all the words starting with 's', then move on to words starting with a 'm' and so on:

./crunch 6 8 smp1abctdkjlhrfgnw2ei0ov3q457z968yux

As a side note, when there is no password creation policy, (aka users are not forced to include numbers in their passwords), incrementing your guesses from right to left is slightly more optimal than password crackers such as Cain&Able who increment their values from left to right, aka 10, 20, 30, 40, etc. The reason for this is because the entropy of the first character of a password is slightly less than the entropy of the last character. For example, from the phpbb.com cracked dataset, over 51% of the users choose one of top ten characters to start their password with. Likewise, 46% of users choose one of top ten characters to end their passwords. It's not a big difference, but every little bit helps. If there is a password creation policy though, this quickly falls apart, and it's vastly more preferable to use a left to right approach since people very frequently put numbers/special characters at the end of their passwords. For example, just the number '1' appears 21% of the time at the end of passwords in the MySpace training list where users were forced to select a password that included non lowercase letters. The results of this test can be seen in Fig. 2.2.1:

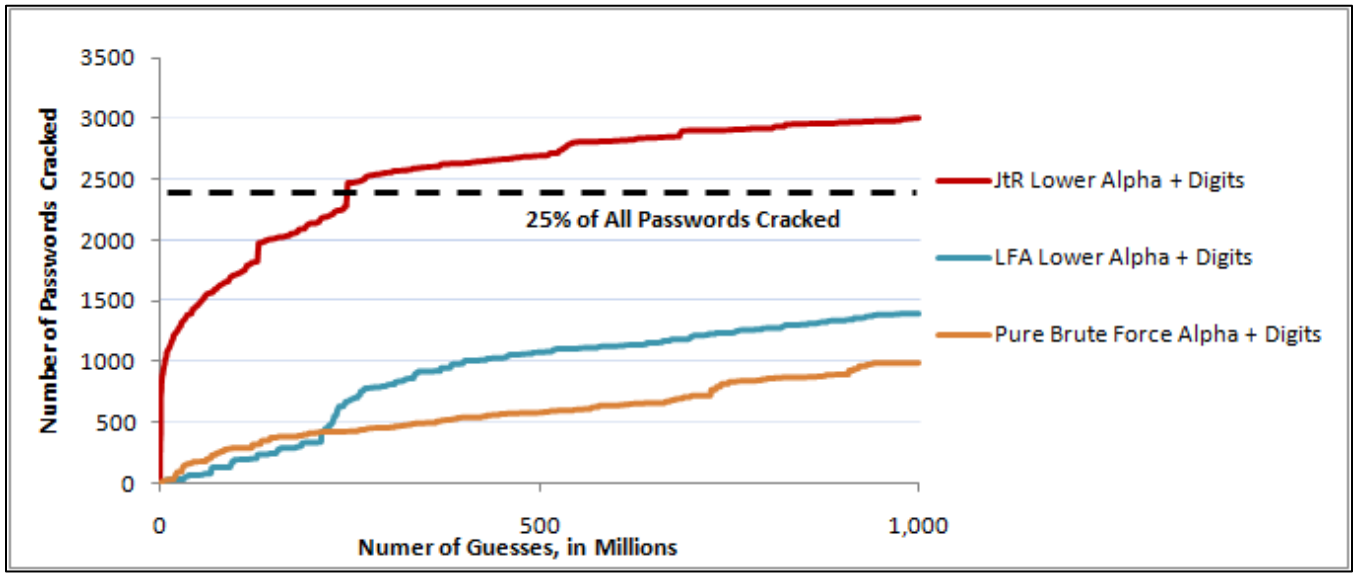


Figure 2.2.1: Comparing a Markov Optimized Attack, Letter Frequency Analysis Attack, and a Pure Brute Force Attack against the Hotmail List

The first thing that stands out about the results is that the letter frequency analysis percentages were not ideal for this test set. It actually performs worse initially then the pure brute force approach.

That being said, given enough time the letter frequency analysis attack still out-performs the pure brute force attack. Comparing the first letter character frequency ordered set of the phpbb list to that generated from the hotmail training list we get:

Hotmail list: `a1mcs2p01betdrjfng3h6kio9v57y48zuwxq`

Phpbb list: `smp1abctdkjlhrfgnw2ei0ov3q457z968yux`

Looking at these frequencies, the best choice would have been to start with the letter 'a' which is coincidentally what the pure brute force approach did. The reason for the huge bump in effectiveness of the letter frequency analysis attack around the 220 million guess mark is because it hit the '1' as the first character, followed by an 'a' which was about the equivalent of a perfect storm since people started their passwords with a number, followed by the actual word they were using. That being said, the results of this test show how superior using a Markov optimized approach is to both a letter frequency analysis and a pure brute force attack. Using less than 40 million guesses, John the Ripper operating in incremental mode managed to crack the same number of passwords as the letter frequency analysis attack did using one billion guesses.

The next question is how these attacks fare against passwords of a set length. The previous test was run against the entire hotmail password list. The distribution of hotmail set by password length can be seen in Fig. 2.2.2.

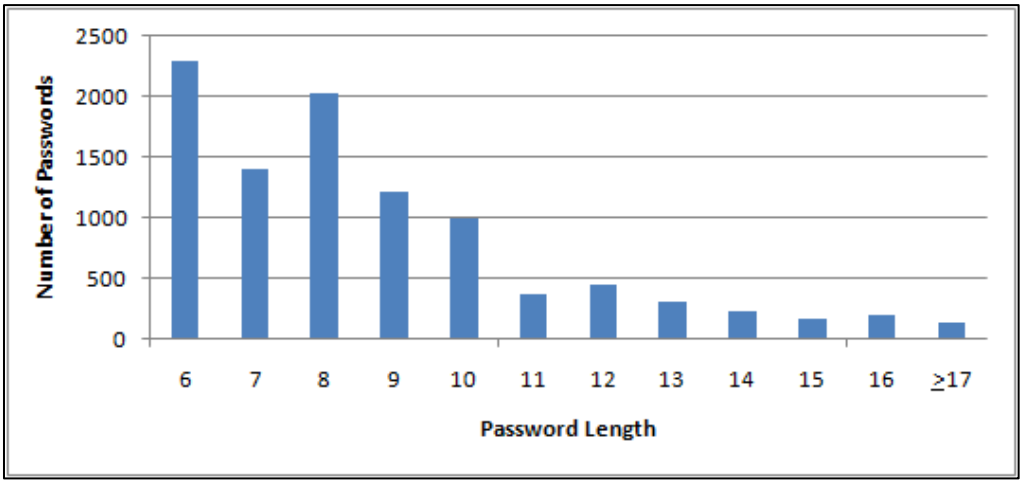


Figure 2.2.2: Password Length of the Hotmail Dataset

To test how these methods perform when cracking longer passwords, the Hotmail password list was subdivided up into three separate lists containing passwords of six, seven and eight characters

long. John the Ripper's incremental mode along with the letter frequency analysis and pure brute force attacks were then run for one billion guesses against each of the subdivided hotmail lists. The only modification from the previous tests was that each attack were tailored to only attempt guesses of the length of the password set they were targeting. Therefore when making guesses against the seven character long Hotmail list, only guesses of length seven were made. The results against six and seven character long passwords can be seen in Fig 2.2.3, and Fig 2.2.4.

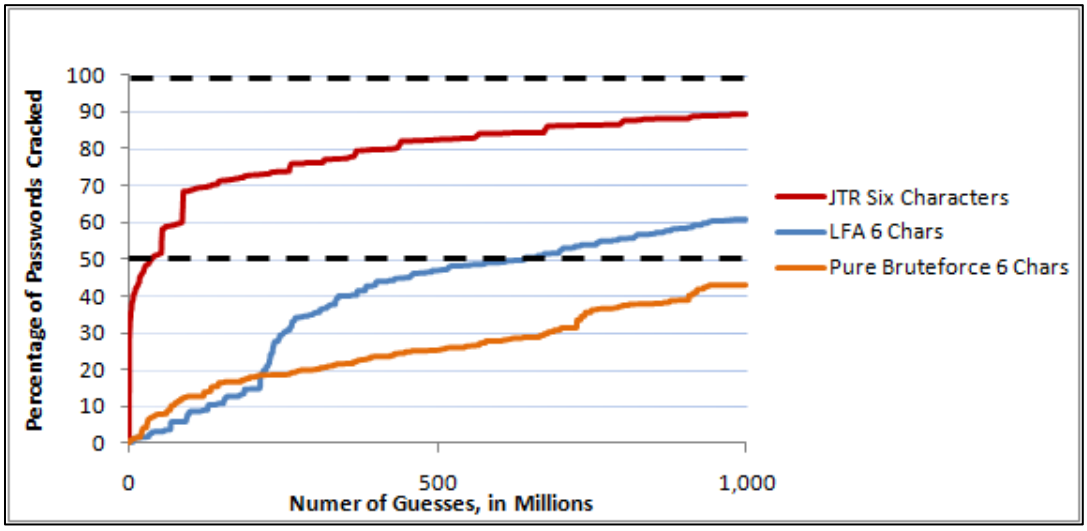


Figure 2.2.3: Various Brute Force Attacks vs. 6 Character Passwords from the Hotmail List

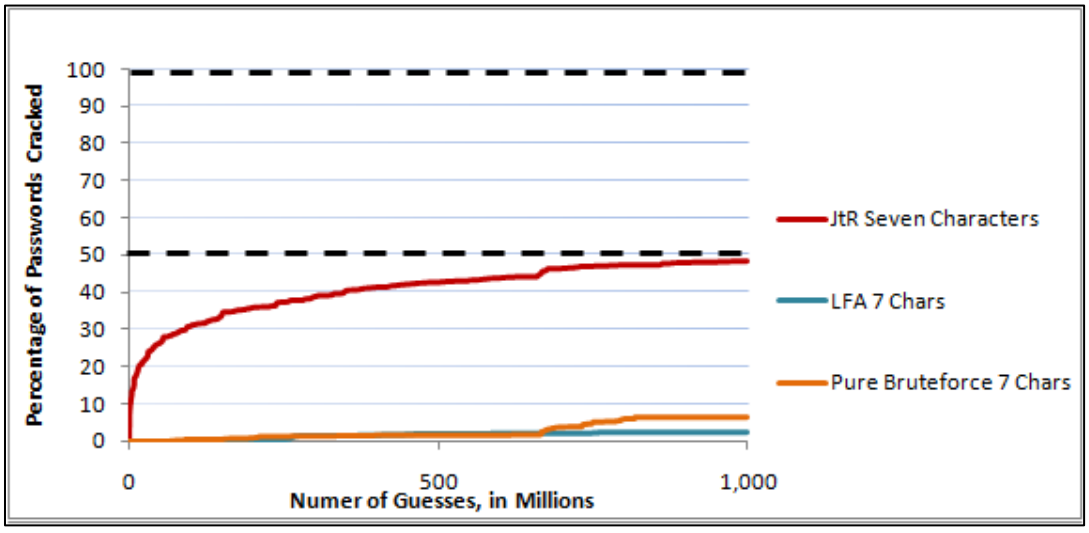


Figure 2.2.4: Various Brute Force Attacks vs. 7 Character Passwords from the Hotmail List

The graph of the attacks against eight character long passwords was not included since it was fairly sparse for the brute force and letter frequency analysis attacks. For eight character long passwords, in the first billion guesses the brute force attack did not crack any passwords and the letter frequency analysis attack only cracked two. This stands in stark comparison to the Markov based attack, where you can see the results of it running against eight character long passwords from the Hotmail list in Fig. 2.2.5 where it cracked slightly more that 40% of the entire set.

In fact, as seen in Fig. 2.2.5, the Markov enhanced attack remains extremely effective even as the password length increases, while pure brute force and letter frequency analysis attacks quickly become infeasible. Even when cracking eight character long passwords, John the Ripper's Incremental mode only required around 185 million guesses to crack 30% of the passwords.

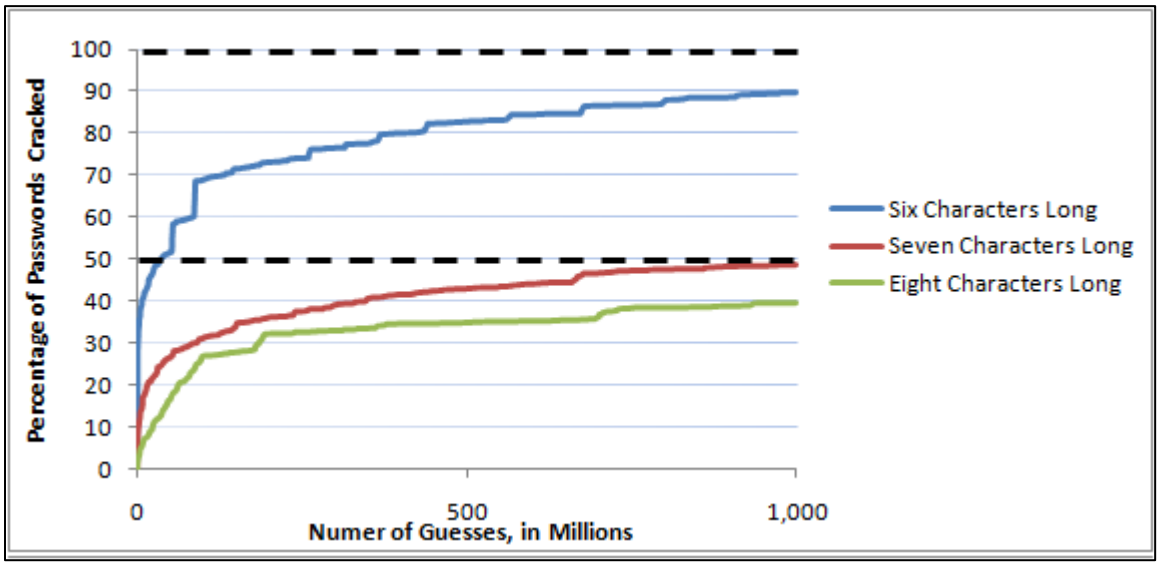


Figure 2.2.5: Markov Enhanced Attacks vs. Various Length Passwords from the Hotmail List

In conclusion, an attacker is just wasting their time if they employ a brute force attack that does not at the very least use Markov probabilities. This can further be enhanced through the use of targeted brute methods to effectively attack longer passwords. Therefore when performing a threat analysis, it is important to use these models. This also shows the applicability of using probabilistic methods to model human password selection. This idea will be further expanded in Chapter 5.

CHAPTER 3

EVALUATING DICTIONARY BASED ATTACKS

3.1 Overview

While brute force attacks were covered in Chapter 2, now it is time to look at dictionary based attacks. In a dictionary attack, the attacker uses a dictionary comprised of words they suspect the target may have used in their password. The attacker then applies mangling rules to these input words, such as capitalizing the first letter, adding two digits to the end, changing the letter 'a' to an '@', etc, to further match the targets password. Because of this, there are three ways that a dictionary attack can fail:

1. The target did not create their password using a method that is vulnerable to a dictionary based attack. As we'll cover in Chapter 3.2, this goes beyond not using dictionary words. For example keyboard combinations are still subject to a dictionary based attack. Instead this category covers techniques like using randomly generated passwords.
2. The base word that the target selected for their password was not in the attacker's input dictionary
3. The attacker did not apply the correct word mangling rules to their input dictionary in order to match the targets password.

The important thing to remember then is that the success of a dictionary based attack depends not only on the input dictionary selected, but also the word mangling rules applied to it. The problem though is that the number of guesses that an attacker can make is finite. This means that the larger the input dictionary the attacker uses, the less word mangling rules they can apply to each individual word. This is because each mangling rule can often create several guesses for each word in the input dictionary. Likewise, if an attacker wants to use highly aggressive mangling rules that can result in thousands of guesses per word, then they must select a highly targeted and very small input dictionary for their cracking session. These two attack techniques, a large input dictionary with simple mangling rules, and a small input dictionary with complicated mangling rules, are not necessarily exclusive. An attacker often makes several runs when attempting to crack a password, first by using a small input dictionary, and then if that attack fails, attempt to crack the password using a much larger input dictionary. Finally if that fails, an attacker can then switch to brute force methods before giving up.

3.2 Creating Input Dictionaries

Finding and creating effective input dictionaries is a non-trivial problem. While many such dictionaries specifically created for password cracking attacks are available online [23, 50], a vast majority of them were created before the year 2001. While numerous base words that people use, such as password and football, remain the same; these dictionaries fail if the target selected a password from more modern cultural references. With the increasing popularity of the internet, along with improved search functionality, several new possibilities for compiling these input dictionaries have become available.

My first such attempt at this resulted in the program WikiGrabber. This is a command line tool that builds custom dictionaries by spidering web pages hosted on Wikipedia. Creating custom dictionaries based on Wikipedia articles actually turned out to be a very difficult problem though as it was hard to construct the appropriate search queries. For example, a search for the term 'beer', would return a page that primarily dealt with the history of the fermentation and brewing of beer. To instead find a page devoted to beer brand names, which would be much more useful in a password cracking session, the attacker needed to search for brewing companies instead.

A much more fruitful endeavor though was to expand the WikiGrabber program to specifically target Wiktionary articles instead. Wiktionary [52] is a sister project of Wikipedia, and is devoted to creating user generated dictionaries for different languages. This offered several advantages when it came to creating custom dictionaries. First, it allows me to create input dictionaries for languages other than English. This is tremendously powerful since there are very few non-English input dictionaries available online. The second advantage is that since the words are categorized by type, (noun, verb, adverb, proper noun, etc), this allows an attacker to create dictionaries ordered by type which can later be used in a pass-phrase cracking attack. A screenshot of the WikiGrabber program can be seen in Fig. 3.2.1.

Figure 3.2.1: Screen Shot of the WikiGrabber Program

The downside of this approach is that these custom dictionaries depend on the level of community support when it comes to building and updating the corresponding Wiktionary pages. Because of that many of the languages listed on Wiktionary only have a few entries. Some of the more widespread languages such as French, German, and Spanish though have much more support. The number of entries for these pages has also grown over time. Consider the wordlist generated from the Finnish Wiktionary entries. When WikiGrabber was first run on it in October 2008, the resulting dictionary contained 96,348 words. When WikiGrabber was run again against the Finnish entries in March of 2010 the new dictionary contained 114,219 words.

This isn't the only example of using Wikipedia sister projects to create input dictionaries. Another dictionary of famous quotes was generated from the website Wikiquote.org. Even when limited to quotes less than 140 characters long, the resulting dictionary contained 187,675 unique quotes to use when cracking passphrases. Probably the ultimate example of mining Wikipedia and its sister projects was taken by research Sebastien Raveau, who created an input dictionary containing every unique word found on every Wikipedia page [54]. The resulting input dictionary just for the English language sites contained close to twelve million words.

Wikipedia is not the only source of input dictionaries. For example, over the course of this research several input dictionaries have been developed to model keyboard combinations such as 'qwerty', and one line ASCII art, such as this picture of a fish '/><{{{{">'. Perhaps the best source for custom input dictionaries though has been from previously disclosed password lists. Using the passwords, 'as is', has been very useful since the datasets available have grown large enough that I now sometimes possess a user's password from a previous discloser. In a more generalized approach though, it has also been useful to lowercase all of the letters in a sample password set and then select all of the alpha strings for a custom dictionary. This is because while the password "tigerwoods1972' might not be that common, the base word 'tigerwoods' is used much more often. This is very helpful since many common words such as 'tigerwoods' do not appear in a majority of the password cracking input dictionaries available online.

3.3 Using Edit Distance to Estimate Coverage

The next question then is how effective are these input dictionaries. What the attacker really wants to know is which input dictionary should they use in their password cracking attack, when should they switch to a larger input dictionary, and when should they resort to brute force guesses. A naïve approach would be to simply run password cracking sessions using different input dictionaries against a common test set, and evaluate the results. The problem with that approach is the amount of time

required to perform such an analysis. Even a simple rule such as add four digits to the end of an input dictionary word can generate ten thousand unique guesses per word that then need to be compared against the test set. Likewise, even though a password created using a word from the dictionary might be in the target set, if the right word mangling rule is not applied during the guessing phase, the password is not going to be cracked.

One possible solution to this problem is to use a modified version of edit distance to evaluate the effectiveness of input dictionaries. The notion of Edit distance has long been used in computer science to determine the number of changes needed to transform one string to another [54]. The goal then is to see if the passwords in the target set can be transformed into a word from the input dictionary within a set number of edits.

While using the edit distance may sound much like running a password cracking attack – both methods attempt to modify a word so it matches a value in the other set – there are important differences. Most importantly, by trying to modify the password to match an input dictionary word, we can safely make some assumptions about the input dictionary since it composition is determined by the attacker. For example, the attacker can convert all uppercase characters to lowercase letters in the input dictionary. Second, in most cases there are usually many fewer passwords in the training set then there are words in the input dictionary. This results in fewer comparisons. Third, unlike with a normal password cracking session, edits do not need to be attempted if they have no chance of matching a word in the other set. For example, in a password cracking attack, an attacker may attempt to add four digits to the end of a dictionary word. If none of the target passwords end with four numbers though, this is a wasted effort.

One model of edit distance is the Levenshtein distance. The Levenshtein distance measures the number of additions, subtractions, or substitutions required to change one string to another. Unfortunately, using a pure implementation of Levenshtein distance would result in a large number of false positives. Consider the password ‘star1234’. It is easy as an observer to tell that it was created by someone adding the number ‘1234’ to the end of the word ‘star’. The Levenshtein distance of this edit would then be four. This can result in false positives though since other words with a Levenshtein distance of four include, ‘stars’, ‘stare’, ‘start’, and ‘starting’. The goal then is to create a modified version of edit distance whose edit rules more accurately model how people create passwords. Examples of the edit rules currently being used by the training program for our password guess generator UnLock, (further discussed in Chapter 5), can be seen in Table 3.3.1.

TABLE 3.3.1: Example Modified Edit Distance Rules

Rule Description	Example Passwords	After Rule Had Been Applied	Custom Cost
Lowercase all letters	PaSSword	password	0
Remove all digits/special characters from the end of the password	password123$	password	0
Remove all digits/special characters from the front of the password	123password	password	0
Perform common letter replacements	p@ssword p@$$word	password password	1 1
Remove the same alpha character from the front and end of a password	xpasswordx	password	1
Remove two of the same alpha characters from either the front or end	xxpassword passwordxx	password password	1 1
Remove common prefix/postfix values	passwords surfing	password surf	1 1
Remove a single alpha character from the front or end of a password	xpassword	password	2
Subtract another dictionary word from the password	coolpassword	cool + password	4

These rules are used with a parser to see if it is possible to transform a password into a corresponding dictionary word within a specified number of edits. Currently the maximum cost allowed is four. Referring to the above table, not all edits are assigned the same cost. That is because some edits are much more likely to cause false positive matches than others. Therefore certain rules such as “lowercase all letters” essentially are treated as free. Other rules though, such as “separate a password into two dictionary words”, are assigned a much higher cost. Besides limiting false positives, another advantage of assigning a limit based on cost is that only a fairly limited number of edits are allowed, which means that time required to parse a dictionary and training list of passwords is usually less than a couple of minutes. For example, using the input dictionary Dic-0294 which contained slightly more than 869 thousand words, and comparing its effectiveness to the MySpace training list containing 33,561 training passwords took less than three seconds on a lab MacOSX 2.2GHz Pentium Core Duo computer. Of course as larger training sets or input dictionaries are parsed, this time increases.

3.4 Results of Using Edit Distance to Evaluate Input Dictionaries

Using the methods described in Chapter 3.3, it's important to see if the metrics derived from using edit distance correspond to results found during a password cracking attack. To that end, the edit distance program was run on several popular input dictionaries. A more detailed description of these dictionaries and how they were obtained can be found in Chapter 5.3.2. The results of comparing the edit distance of these dictionaries to the MySpace training set can be found in Table 3.4.1.

TABLE 3.4.1: Matches between Common Input Dictionaries and the MySpace Training Set

Dictionary Name	Dictionary Size (words)	Percent matched
Dic-0294	869,228	61.04%
Passwords.lst	3,115	24.34%
Common_Passwords	816	6.87%
English_Wiki	68,611	40.01%
Wordlist.txt	306,706	47.96%

The next step was to run a password cracking session against the MySpace test list to see how the edit distance compared to a dictionary based attack. For this attack the password cracker John the Ripper was used [23], operating in wordlist mode with the above dictionaries. For the mangling rules, since the default John the Ripper ruleset is fairly short, the "single mode" wordlist from John the Ripper was selected instead. Each cracking session was then allowed 100 million guesses. That being said, due to the different dictionary sizes, not all of the dictionaries actually used all 100 million guesses before terminating even with the "single mode" rule-set. The results can be seen in Fig. 3.4.1.

The first thing to note is that given even a hundred million guesses, none of the attacks, with the exception of the Common_Passwords list, came close to reaching the percentage given by the edit distance calculation. This is actually to be expected. The metric provided by edit distance ideally should be an upper-bound of the possible effectiveness of a dictionary based attack. Another interesting result was how much more effective the English-Wiki dictionary proved to be compared to the Dic-0294 input dictionary. This shouldn't be surprising though since the English_Wiki dictionary was significantly smaller than Dic-0294 while still managing to match 40% of the passwords using the edit distance metric. Since each cracking session was limited to 100 million guesses, the session using Dic-0294

was never able to run to full completion. Therefore, given that Dic-0294 contained a large number of words that were unlikely to result in cracking a password, it wasted many of its guesses mangling very low probability words.

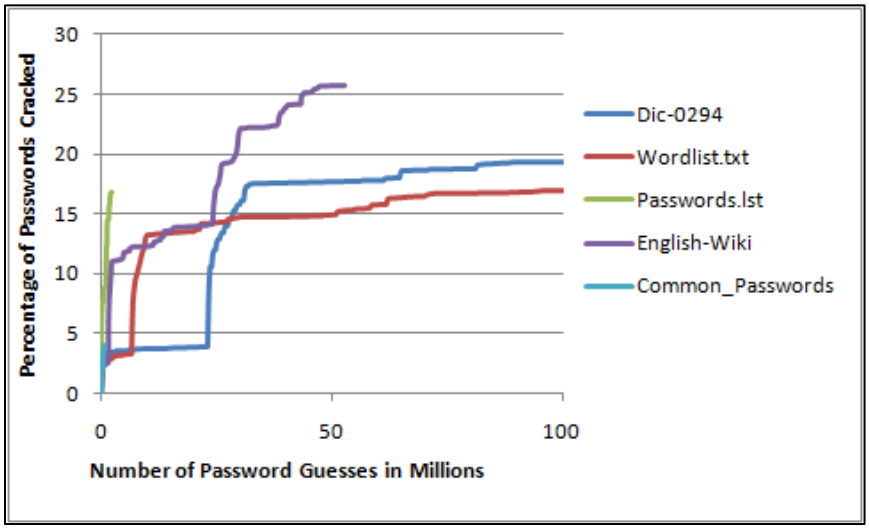


Figure 3.4.1: Different Dictionaries Run Against the MySpace Test List

Given those results, the next question then is how Dic-0294 performs relative to the English_Wiki when both cracking sessions are allowed to run to completion. In addition, to increase the number of guesses generated for this test, the rules for John the Ripper's Single mode that are turned off by default were enabled. The results of this test can be seen in Fig. 3.4.2.

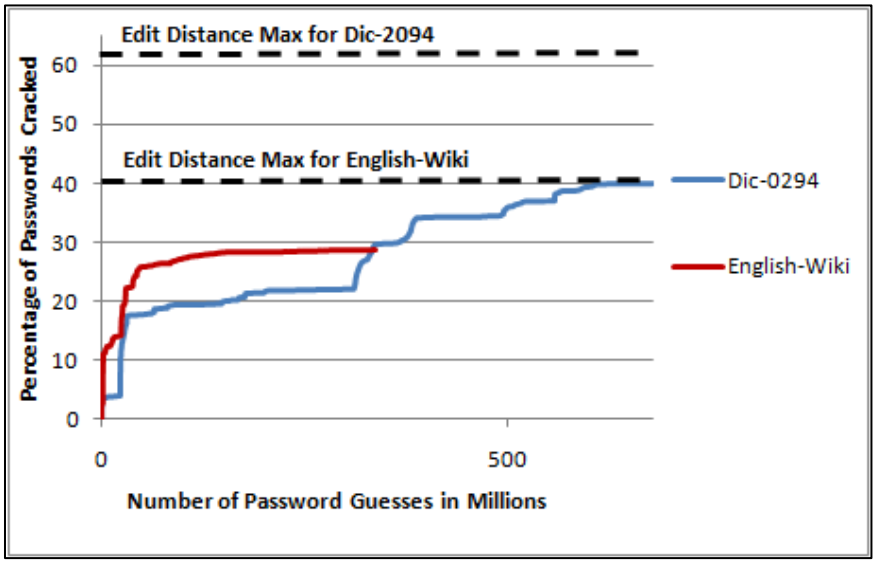


Figure 3.4.2: Longer Cracking Session Run Against the MySpace Test List

As can be seen in Fig. 3.4.2, given more guesses, Dic-0294 eventually cracks more passwords than the English-Wiki input dictionary. It is notable though that English-Wiki cracked slightly more than 70% of the number of passwords that Dic-0294 cracked. The reason why this is significant is because according to the edit distance calculations it was predicted that the English-Wiki dictionary would crack 65.5% of the passwords in the test set compared to Dic-0294. This means in addition to gaining a general understanding of the maximum number of passwords that each input dictionary is likely to crack; the edit distance calculation also gives us a reasonable approximation of how input dictionaries perform in relation to each other in a password cracking session. This matches up with the results in Fig 3.4.1, where the common_passwords input dictionary performed 65% as well as the English-Wiki dictionary, while the edit distance metric predicted it would crack only 60% as many passwords as the English-Wiki dictionary. It's important to note that this relative effectiveness metric is dependent on each password cracking session being run with the same word mangling rules applied to them, and for each session to be run to completion.

This idea of rating the relative effectiveness of input dictionaries prove to be an important component of the probabilistic password cracking program, UnLock, which will be described in more detail in Chapter 5. Even for traditional password cracking attacks though, this metric is useful since when combined with the size of an input dictionary it can help guide an attacker to which dictionaries they should employ when cracking passwords.

While evaluating the effectiveness of input dictionaries is very useful, edit distance can also help identify the word mangling rules used create a password. For example, Table 3.4.2 shows the top ten most common rules, (out of 1,010), found when performing the edit-distance calculation on the MySpace training set using the input dictionary dic-0294

Table 3.4.2: Top 10 Rules Found in the MySpace Training Set Using Edit Distance

Rank	Rule	Number of Occurrences
1	Append two digits	4920
2	Append one digit	4121
3	Append three digits	1420
4	No-op – Do nothing	1152
5	Append four digits	1016
6	Combine two words from the input dictionary	824
7	Append one special character	713
8	Insert one digit in the front	343
9	Insert two digits in the front	230
10	Uppercase the first letter, append one digit	180

Most of the rules listed above, with the possible exception of rule #6, are fairly basic reflecting the reality that a majority of people do not employ complicated mangling rules if they are not required to. Where this approach has merit though is identifying less likely word mangling rules. For a sample of some of the more interesting rules, refer to Table 3.4.3.

Table 3.4.3: Interesting Rules Found in the MySpace Training Set Using Edit Distance

Rank	Rule	Number of Occurrences
12	Lowercase the first character, uppercase the rest, append one digit	139
19	Replace the letter ‘i' with a ‘1’	74
20	Append the letter ‘s’, append a digit	70
21	Append a special character, append a digit	67
28	Replace the letter ‘e’ with a ‘3’	48
29	Replace the letter ‘o’ with a ‘0’	47
30	Append the letter ‘y’, append two digits	47
31	Append the letter ‘y’, append two digits	45
34	Append a digit, append a special character	42
102	Replace the letter ‘a’ with a ‘@’	13

From this we can learn that mangling rules such as lowercasing the first character and uppercasing the rest, while infrequently used, is much more likely to occur than many other mangling rules. Also we can see that switching the letter ‘i’ with a ‘1’ is the most common letter replacement. Likewise, replacing the letter ‘a’ with an ‘@’ occurs very rarely. Also if someone appends both digits and special characters to their password, they are much more likely to append a special character followed by a digit, (aka password$1), instead of the other way around.

The next question is if this knowledge can then be weaponized for a password cracking attack. A naive way to accomplish that would be to create word mangling rules based directly on their occurrences found in Table 3.4.2. Therefore a custom configuration file for John the Ripper was created based on the first ten rules found by the edit distance calculation. The one exception was rule #6 which was omitted due to difficulties modeling it using John the Ripper. Password cracking sessions using the input dictionaries Dic-0294, English_Wiki, and Passwords.lst were then rerun against the MySpace test list with the modified mangling rules applied. Each run was once again limited to one hundred million guesses. The results of these tests, compared to the original cracking sessions shown in Fig 3.4.1, can be seen in Fig. 3.4.3.

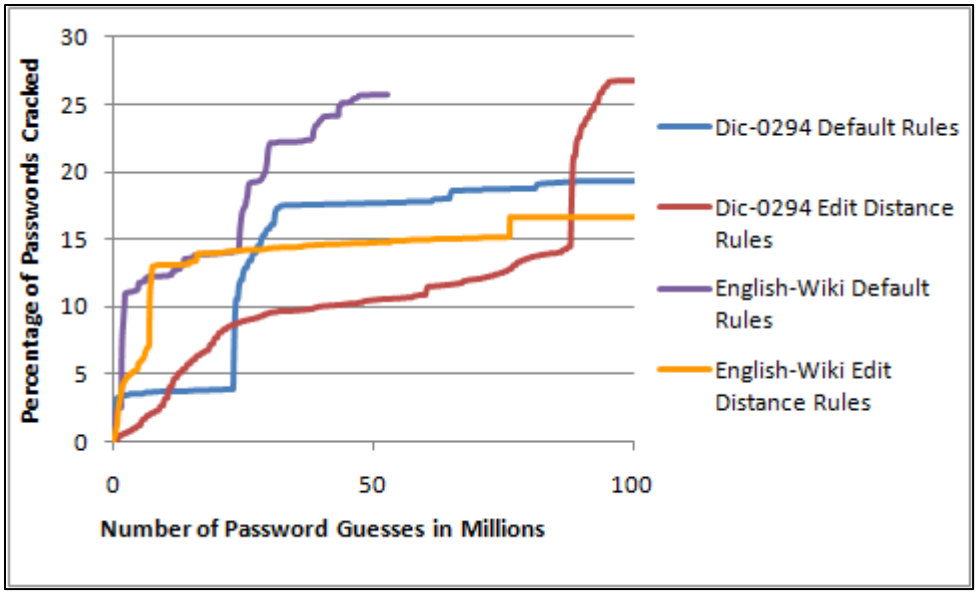


Figure 3.4.3: Comparing Edit Distance Generated Rules vs. the Default John the Ripper Rules on the MySpace Test Set

The results of this run were decidedly mixed, with the edit distance rules performing better than the default John the Ripper rules when dic-0294 was used as an input dictionary, but much worse when the English-Wiki dictionary was used. The reason for this is the above approach does not take into account the number of guesses generated per rule. For example, while appending three digits was slightly more common than not applying any word mangling rules, it would generate one thousand times more guesses as it adds the numbers 000 through 999 to the end of every dictionary word. Also, the naïve way demonstrated is much too coarse in its rule classification. While appending three digits to the end of a password is very common, the vast majority of those three digit numbers are '123'. Ideally you would want to create a separate rule that would just append '123' and leave the other three digits numbers for a later rule to take care of. A must smarter algorithm that can use this data is described in much more detail in Chapter 5: "Using Probabilistic Grammars for Password Cracking".

That being said, the edit distance metric can also be used to classify the word mangling rule applied by a user to create a password. This can be very useful in a password strength checker, as the mangling rule can then be compared to known password cracking rules. From those rules, and common input dictionaries, an estimate can be made on how strong the user's password is. In turn, the resulting measurement can be incorporated into password creation policies that would reject weak passwords.

CHAPTER 4

DICTIONARY BASED RAINBOW TABLES

4.1 Overview of Rainbow Tables

In a standard offline password cracking attack, the attacker possesses a password hash, and is attempting to guess the password that created it. Depending on the hashing algorithm used, the attacker generally spends a vast majority of the cracking session hashing their guesses, (step 2 of a password cracking session as seen in Chapter 1.2). If a password salt is not used though, these hashes generated by the attacker can be stored and reused in future password cracking attacks. This is because the same guess always hashes to the same value.

The simplest way to perform these pre-computation attacks is to create a database of previously hashed guesses and perform lookup on the target hash whenever starting a new password cracking session, as seen in Table. 4.1.1. If the hash is found, the attacker simply finds the corresponding plaintext value stored in the database and then has cracked the hash. Such lookups can be done very quickly, (usually seconds), and drastically reduce the time it takes to crack the most common passwords [10]. Such a database is called a hash lookup table.

Table 4.1.1: Example Hash Lookup Table

Index	Plaintext Value	Hashed Value		Target Hash
1	password1	7C6A180B36896		
2	monkey123	CC25C0F861A83	←	CC25C0F861A83 (Match)
3	F00tball	56D4F3C64A5DB		
4	qWeRtY	87BA00BE932A6		

While hash lookup tables have many advantages, their main weakness is their large file size. A good example of this can be found in the Church of Wi-Fi's WPA/WPA2 hash lookup tables [20]. WPA and WPA2 password hashes are salted with the SSID name and length. Therefore, one set of tables was created with one thousand different SSIDs used as salts and an input dictionary of one million words. This created a total key space of one billion, (1,000 x 1,000,000). To put it in perspective, that is

the equivalent of taking an input dictionary of one million words and adding three numbers to the end of them, aka ‘password111’, ‘password112’, ‘password123’… The total file size of those tables ended up being a little more than thirty three gigabytes. Therefore, by adding even a moderately complicated word mangling rule, the resulting hash lookup tables can easily take up terabytes or more of disk space.

Rainbow tables can be best thought of as a very efficient, but lossy, compression algorithm for hash lookup tables. Being a time-memory-tradeoff algorithm the computational requirements to perform a lookup are higher than with a hash lookup table, but the disk space usage is significantly lower. The basic idea was first proposed by Hellman [34] where chains of password hashes were stored and then recreated to attack password hashes. The idea was further refined by Oechslin [24] who introduced a different reduction function for each step of the hash chain and removed the idea of distinguishing points. A popular implementation of Oechslin’s attack, rainbowcrack, was later coded and released by Shuanglei [25].

4.2 Rainbow Table Construction

An important concept of rainbow tables is the idea of an index value. The index value ranges from 0 to (key-space max -1). To demonstrate this, if the attacker was trying to brute-force all words six characters long which contain only lowercase letters the key-space max would be 26^6. What this means is that each possible value of the index variable represents one unique password guess. Therefore any attack involving rainbow tables must have a fast indexing function.

There are three main functions used in the creation and application of rainbow tables, and they are cyclical in operation. These functions are *IndexToPlain*, *PlainToHash*, and *HashToIndex*. The IndexToPlain function takes an input index value and returns the corresponding plain-text password guess. As the name implies, the PlainToHash function takes an input plaintext password guess and returns the hashed value of it. Finally the HashToIndex function takes a hash as input and makes use of a reduction function to return an index value.

Rainbow tables are based on the idea of hash chains. When a rainbow table is created, the user specifies the chain length, and the number of chains. The chain length value can be thought of as the amount of compression to use. The longer the chain is the more hashes that can be stored in the same amount of disk space. The downside is that a longer chain takes more time to search during an attack and is more prone to collisions. The number of chains determines how many chains are stored, and thus the disk space the rainbow table takes up. When a traditional rainbow table hash chain is created, an initial index value is randomly selected. IndexToPlain is then run to determine the plaintext password

guess. That guess is then run through PlainToHash where it is hashed. Finally HashToIndex is run to determine the next index value. This process, (using the previous index value as input), is repeated **n** times where **n** is equal to the chain length. An example of this can be seen in Figure 4.2.1

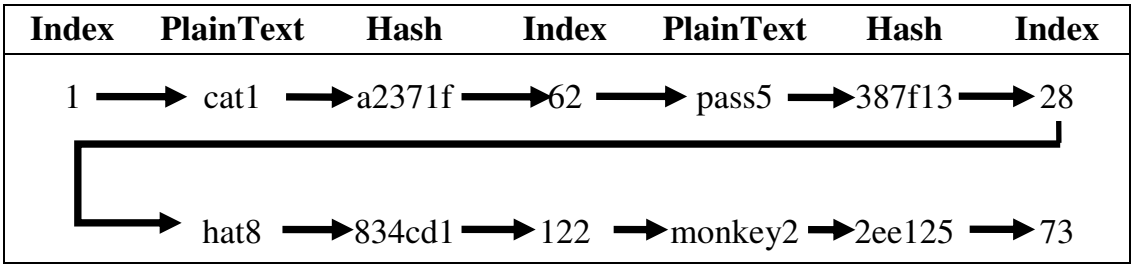


Figure 4.2.1: Creating a Rainbow Table Chain of Length Four

Rainbow tables obtain their compression by only storing the first and last index values for each chain. This means that while the individual chain itself may represent several thousand hashes, only two values are stored. Since the process to create a chain is deterministic, the entire chain can be recreated from the first index value. The last index value is saved so that the target-hash, (the hash the attacker is attempting to crack), can be identified if it exists in that chain.

When attempting to crack a password hash using a rainbow table, the attacker takes the target password hash and then runs HashToIndex on it. They then compare the generated index value to the last index value of every chain in the rainbow table. If the password hash's index value matches one of the chains, the chain is then recreated from its first index value. If the target hash exists in the chain, the attacker simply goes back one step to see the plaintext that created it. For example, suppose the attacker is attempting to crack the password hash '2ee125' using the chain represented in Figure 4.2.1. They would run HashToIndex on the target hash and obtain the index value '73'. The attacker would then compare it to the last index value for each chain they previously created, and see it matches the value in Figure 4.2.1. They would then recreate the chain using the initial index value '1'. Once they see the hash, '2ee125', in the recreated chain, they simply go back one step to find the plaintext value 'monkey2'.

Now this only detects if the target password hash is the last hash in the rainbow chain. To check every position in the rainbow table, the attacker simply continues to build a target chain using the HashToIndex, IndexToPlain, PlainToHash functions for the chain length of the rainbow table. Therefore, after running the first iteration of HashToIndex on the target hash and not finding a match, the attacker would then run IndexToPlain, PlainToHash, and finally HashToIndex again to calculate the next index value to search against the rainbow table. In most cases though, this is more complicated since a majority of rainbow tables created since the Oechslin paper [35] use a different reduction

function depending on where you are in the chain creation process. As stated earlier, the reduction function is used by HashToIndex to determine what the next index value is given a hash value. What this means is that the same hash value may generate many different index values depending on what step it is in the chain creation process. Because of this, a new target-chain must be created for each possible position that the target hash value may occur in the rainbow chain. For example, if we were trying to crack the password hash 'a2371f' using a rainbow table with a chain length of four, we would generate target-chains as seen in Table 4.2.1.

Table 4.2.1 Cracking a Hash Using Rainbow Tables with a Chain Length of Four

Target Hash	Index	Plain Text	Hash	Index	Plain Text	Hash	Index	Plain Text	Hash	Index
a2371f	102									
a2371f	37	rat2	64257f	97						
a2371f	34	bat2	834cd1	107	stuff1	735821	81			
a2371f	62	pass	387f13	28	hat8	834cd1	122	monkey2	2ee125	73

Only the rightmost index value from each target-chain is used to check against the rainbow table chains. Referring back to the chain in table 4.2.1 again, we can see that the hash 'a2371f' as listed in Table 4.2.1, occurs at the very beginning of the chain, and the plain-text value is 'cat1'.

4.3 Rainbow Table Collisions

One of the main problems that plagues the creation and use of rainbow tables is merging chains. A chain is said to merge when two or more chains with different initial index values share the same last index value. This can be caused by a collision generated by any of the three main rainbow table functions, IndexToPlain, PlainToHash, or HashToIndex. A collision is when one of the functions takes different inputs but returns the same output. Merged chains are problematic since all work done in the chain after the point where the collision occurred is duplicated. Also, if a collision occurs between one of the target chains being generated to check a password and a chain in the rainbow table it can create a false alert, otherwise known as a false positive. When this occurs the chain in the rainbow table is recreated even though the target hash is not in it. Therefore, these false alerts can drastically increase the time it takes to crack a password hash.

Depending on the hash type targeted, the PlainToHash function is generally extremely resistant to collisions. While hashes such as MD5 have been shown to be vulnerable to birthday attacks [36], the probability of such a collision occurring during rainbow table creation is limited. Even if a collision does occur from this function it is rare enough not to negatively affect the performance of the rainbow table as a whole. With weaker hashes such as MYSQL323, which is only 64 bits long, collisions occur much more often. If the collision occurs for the target hash, the rainbow table returns a valid password for the user even though the "cracked" password is different from the password the user originally selected. Unfortunately, a vast majority of the collisions will be generated by password hashes not matching the target hash, which results in wasted work and merging chains.

Traditionally the IndexToPlain function has also been resistant, if not completely devoid of collisions. This is due to the nature of brute-force attacks that have been employed. When generating guesses, (as an indexing function), with a brute force approach it is fairly trivial to avoid duplicate guesses, and thus collisions from the indexing function. As we will see later though, when applying a dictionary based approach special care must be taken to avoid duplicate dictionary words as well as rules that generate duplicate guesses. Because of this, it is important to be aware that dictionary based rainbow tables require much more planning in their construction, and the attacker may have to accept a larger amount of collisions in their table than if they used a brute-force approach.

Finally, the HashToIndex function is where most collisions occur. This is due to the nature of rainbow table construction as the goal in creating a rainbow table is to represent as many plain text guesses and thus as many index values from the key-space as possible. Due to the probabilistic nature of creating chains, (aka you have no idea what the index value of the next step is until you create it), an attacker has no way to avoid duplicate guesses generated by the HashToIndex function. To combat this, the HashToIndex function normally uses a different reduction function depending on what step it occurs in the chain. This way, even though duplicate guesses may be generated, unless those duplicate guesses occur in the same step in the rainbow chain the chains themselves do not merge. This limits the damage that is caused by collisions from the HashToIndex function.

4.4 Previous Work

The main difference between most Rainbow Table implementations is their IndexToPlain function. How a Rainbow Table indexes the key-space determines what type of guesses they can make. Previous Rainbow Table IndexToPlain functions worked on variations of brute force type attacks. The first type of attack, popularized by Shuanglei's program rcrack [25], relies upon a pure brute-force selection mechanism. When generating a rainbow table using this approach, a character-set is selected along

with the desired length of password guesses to brute-force. For example, a table could be generated to brute-force passwords of length one through five, that contain letters (upper/lower), and numbers. The key-space max is then determined, in this case $\sum_{k=0}^{5} 62^k$ which equals 931,151,402, and the maximum index then is 931,151,401, (key-space max -1). The IndexToPlain function for these traditional brute-force Rainbow Tables takes an index value and then figures out the length of the password guess by dividing the index value by the size character-set, until the resulting value equals zero. The number of successful divisions is the length of the resulting password guess. From there it is fairly trivial to determine the characters to use via modular arithmetic. A *modified index value* is created by **(index_value – charset$^{(length-1)}$)**. This modified index value is then run through successive rounds where that character at the current position is determined by (**modified_index_value mod charset)** and then the new modified index value equals (**modified_index_value div charset**).

Using the above example, in which the character set contains 62 characters, (lower/upper alphas, and numbers), and the input index value 388, the output password guess would be two characters long, **(388/62= 6, 6/62=0)**. The first modified index value would then be **(388 - 64$^{(2-1)}$=326)** which equals 326. This makes the first character a 'q', **(326 mod 62 = 16 = 'q')**. For the next character, the modified index value would then be five, **(326 div 62 = 5)**, which would cause the second character to be 'f', **(5 mod 62 = 5 = 'f')**. Therefore the outputted result of running IndexToPlain on the input 388, would be the password guess 'qf'.

The main problem with the above approach is that it quickly becomes infeasible to brute force the entire key-space for longer passwords. One attempt to help mitigate this problem is to only try to search a subset of the key-space by performing a targeted brute force attack. This approach was popularized by the program rcracki [26]. In this case, the targeted brute-force attack lets the attacker specify a unique character set to use for each letter position. What this means is that an attacker can generate guesses following rules such as "brute force all passwords that start with four characters, followed by two numbers." Example guesses generated by this rule would be aaaa01, aaab02 … zzzz99. This lets the attacker tailor their approach to target certain user password selection characteristics, such as users generally put numbers at the end of their passwords. By using this method, these tables can attack a majority of human generated passwords while limiting the key space the attacker needs to search. The actual algorithm used is much like the pure-brute force approach described above, but the character set is determined based on the current position being filled in by the IndexToPlain function.

A third way to implement the IndexToPlain function to make use of Markov models was suggested in the Narayanan paper [19]. The Narayanan approach first generates a dictionary of all Markov guesses whose probability falls above a specified limit using the algorithm previously described

in Chapter 2.1.3. If no other mangling rules are applied, then the IndexToPlain function simply indexes into the generated results to create a password guess. As you can imagine, this is highly inefficient since the Markov generated dictionary contains every plain-text guess, and in this case a hash lookup table would be much more effective. They solve this problem though by adding a second dictionary that applies word mangling rules, (such as capitalization, and adding digits to the end of a password guess). In this way, their attack becomes a targeted brute-force attack. The second dictionary's rules are applied by finding an index into the Markov generated input dictionary, along with an index into the rules dictionary. This secondary rules dictionary is used much like a mask that is then applied to the Markov guess. This is done using the following function get_key5 [19]:

```
get_key5(index)
{
        count1 = partial_size1(0, 0)
        count2 = partial_size2(0, initial_state)
        index1 = index/count2 // quotient is truncated
        index2 = index - index1 * count2
        key1 = get_key1(0, index1, 0)
        // we assume that key1 consists of
        // lowercase characters
        key2 = get_key2(0, index2, initial_state)
        key = ""
        pos = 1
        for char in key2:
                if char is 'a'
                        append key1[pos] to key
                        pos = pos+1
                if char is 'A'
                        append uppercase(key1[pos]) to key
                        pos = pos+1
                if char is neither 'a' nor 'A'
                        append char to key
        return key
}
```

While the above algorithm does allow the use of mangling rules, it still has several major weaknesses. The main problem is that the secondary dictionary, containing the mangling rules, must contain a separate entry for every case mangling rule, and more importantly, digit added. Therefore, if an attacker wanted to add four digits to the end of each Markov generated word, they would need ten thousand entries in their secondary dictionary, representing 0000 to 9999. This gets worse when case mangling is applied, as a separate entry must be created for each combination of case mangling rule

and digit appended or pre-pended. The second problem is it does not support additional mangling rules, such as letter replacements, (aka replace ‘a’ with an ‘@’), or rules such as combine two Markov generated words.

4.5 Indexing Dictionary Based Rules

4.5.1 Overview of Drcrack

The Rainbow Tables mentioned before all were variations of brute-force style attacks. Part of my research was to develop a way to store pre-generated dictionary based attacks in a Rainbow Table. This can be thought of as a more general implementation of the Narayanan method [19]. The goal was to develop an IndexToPlain function that:

1. Supported arbitrary input dictionaries, such as those composing human generated dictionary words, dictionaries containing common keystroke combinations, Markov generated input dictionaries, etc.
2. Allowed a vast majority of mangling rules to be applied to each input dictionary word
3. Minimizing the file-size of the overhead associated with the finished Rainbow Tables. This means the previous method used in the Narayanan paper of using a mask to represent each word mangling rule was not suitable.
4. Given an input index number, generate the resulting plain-text value as quickly as possible.
5. Minimize the number of collisions generated by the IndexToPlain function.

The end result was an indexing function that works with a majority of word mangling rules, with it being implemented and publicly released in the tool drcrack [37]. Drcrack gets its name because it is a dictionary based version of the rcrack Rainbow Table creation tool, (aka d-rcrack). The actual Rainbow Tables created by drcrack are composed of four separate files. The first file is the configuration file. This file holds information about the other three files, (such as file names), version information, and global information, such as the Rainbow Table chain length, and number of chains. The second file is the actual Rainbow Tables themselves, which are the first and last index values from every chain in the rainbow table. The third file is the input dictionary used. The last file contains the word mangling rules used, which look much like the word mangling rules used in password cracking programs like John the Ripper [23].

The key was to generate an efficient indexing function for each type of word mangling rule. A user creating a rule-set for drcrack assembles their rules using the GUI frontend shown in Fig 4.5.1.1.

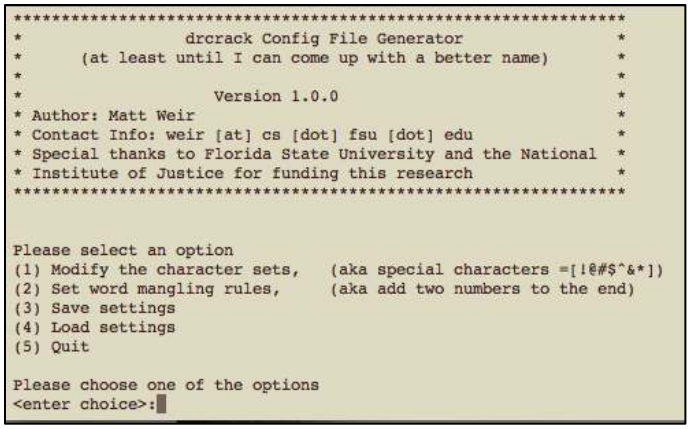


Figure 4.5.1.1: Screenshot of the Drcrack Rule Configuration Generator

Option 1 allows the user to specify global rules, such as character sets allowed. This is most frequently used to specify character groups for insertions, such as which special characters, (aka !@#$%^), to append to a password guess. Option 2 is where the user composes word mangling rules. They do this by combining base word mangling rules together as can be seen in Fig. 4.5.1.2:

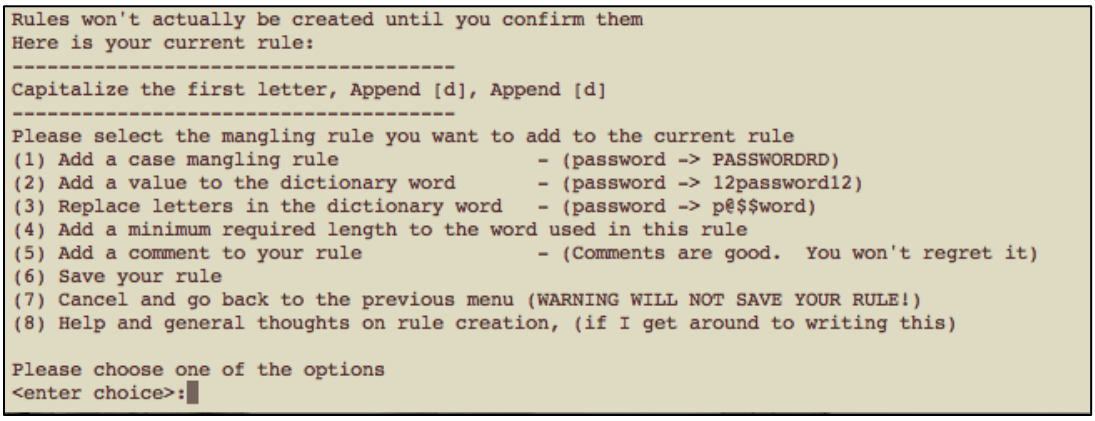


Figure 4.5.1.2: Creating a Rule with Drcrack

The rule currently being worked on in Fig. 4.5.1.2, which can be seen near the top of the screenshot, actually is composed of three base rules:

1. Capitalize the first letter
2. Append a digit
3. Append a digit

Therefore the displayed rule creates guesses such as ‘Password00’ through ‘Password99’. Multiple rules may be contained in a single Rainbow Table. As an example of that, you can see in Figure 4.5.1.3 the first fourteen rules, (out of fifty-five), included in one of the custom Rainbow Tables generated for drcrack. This table is currently also available online for public use [37].

```
------------------------------ALL CURRENTLY CONFIGURED DRCRACK RULES IN ORDER------------------------

-----NOTE, FOR A SLIGHT PERFORMANCE INCREASE IT HELPS TO HAVE RULES WITH THE BIGGEST KEYSPACES FIRST----
(1) Append [d][d][d][d]

(2) Capitalize the first letter, Append [d][d][d][d]

(3) Append [s][d][d]

(4) Capitalize the first letter, Append [s][d][d]

(5) Append [d][d][d]

(6) Capitalize the first letter, Append [d][d][d]

(7) Prefix [s], Append [s]

(8) Append [s][d]

(9) Append [d][s]

(10) Capitalize the first letter, Append [s][d]

(11) Capitalize the first letter, Append [d][s]

(12) Replace 'a' with '@', Append [s][d]

(13) Replace 'e' with '3', Append [s][d]

(14) Replace 'i' with '1', Append [s][d]
```

Figure 4.5.1.3: First Fourteen Rules of the NTLM_Basic Table

Each rule has a corresponding Index size, (the number of possible guesses it represents), and can be thought of as a unique container. When IndexToPlain is called, it first finds which container the index value falls in. Currently this is done via a sequential search, (ordered by the container size), since certain rules generate many more guesses than others. It would be trivial though to instead modify the algorithm to perform a binary search if there were a large number of equally sized rules. Once the target container is known, a modified index value is generated by subtracting the container’s start location from the index value. This simply gives an index location for that particular rule that is independent of all the other rules. The rest of the indexing function depends on the type of mangling rules applied and will be described in more detail in the next several sections.

4.5.2 Performing a Plain Dictionary Based Guess

The first rule type is a simple No-op. While many other rules are applied in a normal cracking session, the No-op is usually also included so that the plain words in the input dictionary can be tried as well. The indexing function for the No-op is straightforward with the Index size equaling the number of words in the input dictionary with a direct one-to-one correlation between the index value and dictionary word selected. The No-op is also often used as the basis for other word mangling rules, such as appending a value, so most rules start with an implicit No-op.

4.5.3 Appending/Pre-pending a Value

The indexing algorithm for appending and pre-pending values is essentially the same, with the only difference being if the value is added to the beginning or end of the dictionary word. For this I borrowed heavily from indexed approach in rcracki [26], where different character sets are used to create a targeted brute-force attack. With drcrack though, instead of a brute-force attack, it regards the input dictionary as one character-set, and the values to append as another character set. Therefore the guess ‘monkey1$’, would be represented the same way rcracki would regard the three letter guess ‘abc’, with ‘monkey’ representing the first character, ‘1’, representing the second character, and ‘$’ representing the third character. To avoid confusion, the term ‘position’ will be used instead of ‘character’. Aka, the guess ‘monkey1$’ would have three positions. The indexing function in drcrack for appending values then works much like rcracki as seen in Chapter 4.4, appends characters with it using remainder of the modified index to decide the value for each position that is currently being referenced. The advantage of this is that is it works independent of all the other rules, so appending a digit and then a special character, is no more difficult then appending four digits. Also other rules as discussed later may also be easily incorporated by assigning their index values to their own position. Therefore, if a previous rule would create ten million guesses, the first position would have a total index size equal to ten million.

Another advantage of this approach is that the user is not limited to only appending or pre-pending individual characters. Dictionary words may also be appended. The best example of that would be the keyboard based dictionary rainbow table created for drcrack. The input dictionary is a collection of 658 common keyboard combinations, such as ‘qwerty’, and ‘1qaz’. It then appends these dictionary words to each other, created guesses such as ‘qwerty1qaz’. While an input dictionary containing 658 words may seem small, by appending the words to each other the resulting key-space can be quite large as seen in table 4.5.3.1:

Table 4.5.3.1: Index Size Created by Appending Keyboard Combinations to Each Other

Number of Appends	Example Guess	Index Size
1	qwerty	658
2	qwerty1qaz	432,968
3	qwerty1qaz2wsx	282,292,528
4	qwerty1qaz2wsx#EDC	185,748,483,424

In the above case, even though a very small input dictionary was used, the resulting Rainbow tables would cover a key-space of over 186 billion guesses. This means extremely large Rainbow tables may be generated with very low overhead for storing the dictionary and related files such as the word mangling rules used.

4.5.4 Case Mangling Rules

While it may seem straight-forward, case mangling proved a unique challenge. The problem is that it can't be expected that all dictionary words are the same length. While rules such as capitalize the first letter may safely be applied to the entire dictionary, what about rules such as capitalize the 7^{th} letter? To that end, when drcrack is run, sub-dictionaries are formed based on word length. These dictionaries are pointers to the main dictionary, but they allow fast indexing for all the words of length **n**. This means the rule, capitalize the 7^{th} letter is only applied to a subset of the total dictionary containing words longer than seven characters long. The key-space referenced represents the sub-dictionaries covered by the current capitalization rule. For individual capitalizations, (aka capitalize one letter), this means the indexing algorithm is treated exactly like the No-op rule, with the exception being that the resulting guesses have their characters at the specified location converted to uppercase.

Multiple capitalizations may also be applied the same way, with the highest index position determining the sub-dictionary used. For example, if two rules such as capitalize the first letter, and capitalize the 9^{th} letter are applied, the entire rule is only applied to dictionary words of length nine or longer. While not implemented in the current version of drcrack, rules could also be created to capitalize two or more arbitrary characters by creating sub-rules of all possible combinations. A more elegant solution might be possible by creating an index function for each sub-dictionary concerning the number of capitalizations.

One possible problem though is since drcrack works with user-defined input dictionaries and rules, the possibility for collisions generated by the IndexToPlain function is particularly high. Therefore many special cases had to be considered when creating an indexing function that involves case mangling. Some of the special cases include:

1. What happens when a character is already uppercased at the location specified? For example the rule is to capitalize the first letter, and the dictionary word is ‘Password’.
2. What happens when a non-alpha character is at the location for the capitalization rule? For example the rule is to capitalize the first letter, and the dictionary word is ‘123456’.
3. What happens when the rule is to capitalize every character and the dictionary word is only one character long, such as ‘a’? This could create a collision with another rule such as capitalize the first letter.

Currently the standard method to deal with the above cases is to generate a unique error return value for the IndexToPlain function. The important part is that the generated error message must be unique from all other return values for the IndexToPlain function to avoid collisions generated by the Rainbow Table algorithm. This means while the resulting guess and hash are essentially wasted, (since the error value is hashed by the PlainToHash function), at least it does not create a collision which might potentially cause several chains to merge.

4.5.5 Letter Replacement Rules

Letter replacement rules, much like capitalization rules, presented a bit of a challenge to implement. Luckily the solution was very similar with the creation of a sub-dictionary for every letter replacement rule currently being used in the created Rainbow Table. This is possible since all of the letter replacement rules being applied to the various rules are known during creation time, and thus drcrack can create the sub dictionaries before it starts generating the tables. For example, if there were two rules, the first being replace all ‘a’ characters with an ‘@’, and in the second rule it said to replace all ‘a’ characters with an ‘@’ and all ‘e’ characters with the number ‘3’, then two sub-dictionaries would be created. The first sub-dictionary would contain all words with the letter ‘a’ in them, and the second sub-dictionary would contain words that had both the letter ‘a’ and the letter ‘e’.

The reason why the sub-dictionary must contain all of the replacements in the particular rule is once again to prevent collisions from occurring. For example, using the above two rules, it’s important

that the second rule is only applied to words with both ‘a’ and ‘e’ in them. Otherwise words such as ‘password’, would be used to generate the guess ‘p@ssword’ twice, would create a collision.

4.6 Results

It is hard to make a direct comparison between traditional brute force rainbow tables and dictionary based rainbow tables. These two attack types are complimentary, just like how brute force and dictionary based attacks are both commonly used together in traditional password cracking sessions. Brute force remains extremely effective due to the widespread use of short passwords composed of a limited character set. Referring back to the real life password sets detailed in Chapter 1.6, take for example the RockYou training list. In that set, over 90% of the passwords contained only lowercase letters and/or digits. Likewise, the average password length was 7.88 characters long. This means a vast majority of these passwords could be cracked via brute force methods. For comparison, some of the free brute force NTLM hashed rainbow tables available on freerainbowtables.com [26] can be seen in Table 4.6.1. These tables were generated via a community based distributed computing collaboration.

Table 4.6.1 Publicly Available Rainbow Tables for the NTLM Password Hash

Password Length	Character Set	Key Space
1-6	All printable ASCII characters	≈ 742 billion
1-12	numeric	≈ 1.1 trillion
1-8	Lowercase alpha, numeric, space	≈ 3.6 trillion
1-7	Lowercase & Uppercase alpha, numeric, space	≈ 4.0 trillion
1-7	Lowercase alpha, numeric, symbol32, space	≈ 7.5 trillion
1-9	Lowercase alpha, space	≈ 7.9 trillion

The above rainbow tables each have an estimated coverage of over 99% of the given key space. So let’s consider the RockYou password list where 64% of the passwords in the training set contain only lowercase letters, digits, and were eight characters long or less. This means a vast majority of those passwords in turn would be broken via a brute force attack by just the third table listed above. Even worse, slightly more than 79% of the passwords in the training set would be vulnerable to at least one of the rainbow tables detailed in Table 4.6.1.

Therefore brute force tables are extremely effective if no password creation policy is in place. On the other hand, if a password creation policy existed requiring each password to contain at least one lowercase, uppercase, number and symbol, and be seven characters or more, none of the passwords from that set would be vulnerable to any of the above brute force tables. Likewise if an attacker wanted to crack more than 79% of the passwords from the RockYou list they would also have to resort to other means besides brute force. That's where dictionary based rainbow tables are useful. They can target stronger passwords that are not vulnerable to brute force attacks. That being said, their effectiveness depends entirely on the input dictionary selected and the word mangling rules used. The fact that they are rainbow tables only means that these dictionary attacks can be pre-computed, allowing an attacker to use larger input dictionaries, and more complicated word mangling rules.

To compare the effectiveness of the described dictionary based rainbow tables, a test set of one hundred randomly selected MD5 password hashes was chosen from the PhpBB test list described in Chapter 1.6.2. The reason only one hundred hashes were included was because the time it takes to perform a rainbow table lookup is multiplied by the number of hashes the attacker is attempting to crack. Therefore, if a lookup takes on average twenty minutes, then cracking one hundred hashes would take approximately thirty three hours to perform. For the dictionary based rainbow table, the input dictionary dic-0294 [50] which contains over 869 thousand words was used. Twenty seven mangling rules were then applied to this input dictionary. These rules ranged from adding four digits to the end of every dictionary word, to capitalizing the first letter and changing the letter 'e' to the number '3'. The resulting rainbow table covered a key-space of over 24.7 billion guesses. The total file-size of the finalized rainbow table was around 456 Megabytes. Some of the other generation data can be seen below:

Generation Info for the Basic MD5 Dictionary Based Rainbow Table

- Hash: MD5
- Dictionary: Dic-0294
- Rules: 27 Mangling Rules – Filename: first_big_test
- Chain Length: 2400
- Chain Count: 5000000
- Number of Sub Tables: 6
- Key Space Covered: 24,792,961,721
- Total Size: 456 Megabytes

To contrast the compression that the rainbow table algorithm provides the attacker vs. a traditional hash lookup table, consider the very popular WPA/WPA2 hash lookup table produced by the Church of Wifi [20]. It only covered a key-space of one billion guesses, yet took up thirty three gigabytes of space. This means that the above dictionary based rainbow table covered twenty four times as much key space while taking up only roughly 1.3% of the disk space. That is the advantage of using the rainbow table compression algorithm.

For the traditional brute force style rainbow table I selected the popular MD5 table from freerainbowtables.com [26] that targeted passwords ranging in length from one to seven characters long, containing lowercase letters, numbers, and special characters. This table covered a key-space of 7.5 trillion guesses and takes up 52.5 gigs of disk space. The generation info for this table can be seen below:

Generation Info for the Brute Force MD5 Rainbow Table

- Hash: MD5
- Character Length: 1-7
- Character Set: loweralpha-numeric-symbol32-space
- Chain Length: 10,000
- Chain Count: 67,108,864
- Number of Sub Tables: 81
- Key Space Covered: 7,555,858,447,479

The results when both tables were run against the 100 test MD5 password hashes from the PhpBB.com list can be seen in Table 4.6.2. The dictionary based rainbow table took around 1/3 the amount of time as the brute force table to finish, and cracked 31% of the total passwords compared to the brute force table's 43% success rate. The most interesting factor was that the percentage of the key-space covered due to false alarms, (aka collisions), in the dictionary based attack was roughly 0.93%, while the dictionary based table covered only 0.01% of the key-space due to false alarms. There actually are several factors that could have contributed to this besides the differences in the two approach's IndexToPlain functions. For example, each of the tables search different sized key-spaces, have different chain lengths, etc. This means that variation in number of collisions may also be caused by the HashToIndex function. That being said, the results certainly point to the fact that even with certain problem cases being taken care of, the dictionary based IndexToPlain function still was generating collisions. Even with the increased number of collisions, and considering the overhead associated with all rainbow tables though, the dictionary based rainbow tables resulted in being able to

search the targeted key-space while calculating only 5.4% of total hashes during the lookup for a hundred different password hashes. When the same attack was run against just one password hash that was not found during the cracking session, the entire lookup with the dictionary based table took less than one minute and calculated only 0.06% of the total hashes from the search space. Therefore, even with the increased number of collisions, a dictionary based rainbow tables can drastically speed up a password cracking attack.

Table 4.6.2: Results of Rainbow Table Attacks against 100 PhpBB MD5 Password Hashes

Value	Dictionary Based Table	Brute Force Table
Passwords Cracked:	31%	43%
Total Cryptanalysis Time:	64.4 minutes	2.93 hours
Total Chain Walk Steps:	1,340,767,315	2,409,386,122
Total Chain Walk Steps Due to False Alarms	233,949,227	1,448,442,733
Total False Alarms	291,636	1,554,388
Percent of the Key-space Covered Due to False Alarms	0.93%	0.01%

But the question remains, is the dictionary based IndexToPlain function the cause of those extra collisions? To that end, I generated one brute force table using the same chain length and chain count as the dictionary based table. For the character-set I selected 30 characters, (a-z, 0-3), and specified for the table to check seven character long passwords. This resulted in a table covering a key-space of 21 billion possible guesses, which is close to the 24 billion possible guesses covered by the dictionary based table. Due to time constraints though, I only created one sub table, vs. the six sub tables created for the dictionary based approach. Since the test is to determine the number of collisions, and not the number of passwords cracked, comparing only one sub-table from each approach did not pose a problem. When both tables were run against one unbroken hash, (that way neither table would crack it and finish early), the dictionary based rainbow table encountered 651 collisions resulting in it generating 2.8 million unnecessary hashes to check the false alarms. The brute force table on the other hand encountered 636 collisions which also resulted in it generating 2.8 million unnecessary hashes to check the false alarms. When run on several other hashes, the brute force table averaged 641.8 false alarms per target hash, while the dictionary based approach averaged 650.8 false alarms per target hash.

While the number of collisions was higher for the dictionary based rainbow table, it was not nearly as significant as indicated by the test shown in Table 4.6.2.

The next question focuses on the effectiveness of a dictionary based rainbow table attack. Since the entire PhpBB list was hashed, the plaintext values are only available for the passwords I managed to crack over a several month long period of time, (See Chapter 1.6.2 for more information). Of the randomly selected 100 passwords used in the above tests, the plaintext values for 97 of the passwords were known. It can safely be assumed that the remaining three unknown plaintext values would not have been cracked by either rainbow table. Manually comparing the plaintext values of the 97 known values to the passwords the rainbow tables did crack, the dictionary based rainbow table cracked all of the passwords that its rule-set and input dictionary targeted. For the traditional brute-force rainbow table, it cracked 43 of 44 passwords that were covered by its key-space. While this falls below the expected 99.92% coverage for that particular rainbow table, statistically speaking it still matches up, as additional tests would need to be run to show if it does not meet the expected coverage. This means that while the brute force rainbow table managed to crack more passwords, that was due to the fact that it covered a much larger search space. It also was caused by weaknesses in the input dictionary and mangling rules selected. For example, Table 4.6.3 shows the analysis of why the dictionary based rainbow table did not crack the sixty six other known plaintext values.

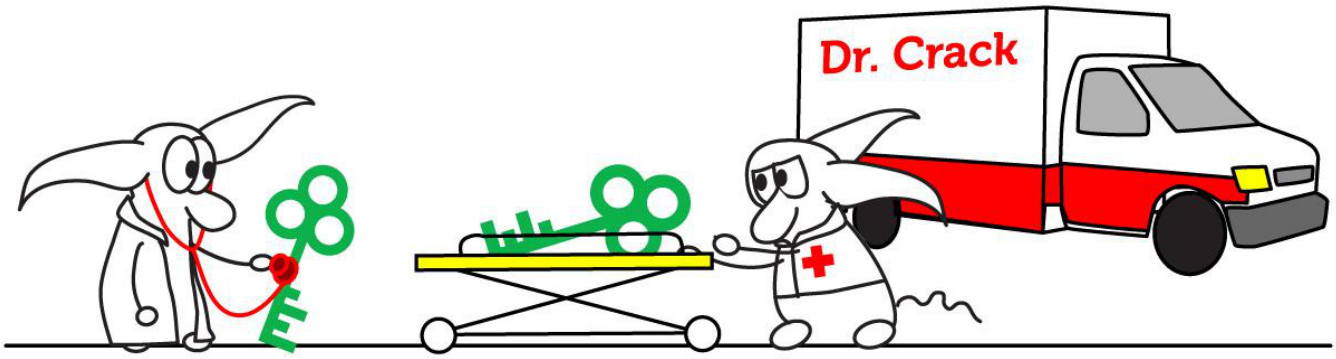
Table 4.6.3: Reasons Why the Dictionary Based Table Did Not Crack Passwords from the PhpBB List

Reason:	**Number of Occurrences**
Base word was not in the input dictionary	27
Correct mangling rule was not applied	9
Both the base word was not in the input dictionary and the correct mangling rule was not applied	7
The target password was a number	11
The target password was a keyboard combination	1
The target password was created via an unknown method	11

As can be seen, a vast majority of the passwords were not cracked due a lack of the base word appearing in the input dictionary Dic-0294. A more comprehensive input dictionary would help reduce this problem. Since the rainbow tables are pre-computed, this actually is a feasible solution, as a larger input dictionary would not dramatically impact the time required to crack a password hash. That being

said, I was unable to classify how a total of 11% of the known plaintext values, or 14% of the total values, were created. Some of these may have been created by using the first letter of each word for a passphrase. Others may have special significance to the person who created them. Unfortunately, unless a model can be generated to target the way that these passwords were selected, they would only be vulnerable to a brute force style attack.

In conclusion, the advantage of allowing rainbow tables to be created using a dictionary based attack plus mangling rules is that it allows an attacker to search a much larger key-space then they normally would using a non brute-force approach. Since rainbow tables are pre-computed, cracking an individual password hash can be done very quickly using a combination of traditional brute force rainbow tables and these new dictionary based rainbow tables. One specific application of this is law enforcement officials need the ability to crack a Windows NTLM password hash in under an hour while they are still performing data collection. This way they can confront the suspect directly with the cracked password in an attempt to have the suspect voluntarily divulge other passwords. Dictionary based rainbow tables make cracking that NTLM password within an hour much more feasible.



CHAPTER 5

USING PROBABILISTIC GRAMMARS FOR PASSWORD CRACKING

5.1 Overview of Probabilistic Password Cracking

The main question in password cracking is how to create your password guesses. Traditionally, as seen in Chapter 3, dictionary based attacks relied upon word mangling rules to generate these guesses. An attacker would attempt to build rules that best mimic how people create passwords in real life. For example, in Table 5.1.1 you can see the first ten default rules that the password cracker John the Ripper [23] uses in its dictionary based attacks:

Table 5.1.1: First Ten Word Mangling Rules in John the Ripper

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

One thing that immediately pops out is Rule #8, which lowercases and reverses pure alphabetic words. As seen from the training sets analyzed in Chapter 3, such a rule wouldn't be that effective as people rarely reverse words when creating their passwords. Yet that rule is run before many other rules which would likely be much more successful at cracking user passwords. This stems from a greater problem in that these rules are generally formed in a fairly ad-hoc basis, based on the attackers'

personal experiences. Ideally we would like to apply machine learning techniques to learn these rules directly from a training set of previously cracked passwords instead.

Another problem with rule based word mangling is that it becomes very hard to incorporate all of the knowledge we have about how people create passwords into an actual password cracking session. For example, some words such as 'password', 'football', and 'monkey' are used much more often as the base word in a user's password than other words like 'zebra', or 'xylophone'. People tend to capitalize the first letter and add digits to the end of their password guesses. Certain digit and special character combinations, such as dates, zip codes, keyboard combinations, etc, are used very frequently. While you see many traditional password cracking programs try to take advantage of this, they do so with only varying degrees of success.

Let's look at the case of trying to conduct a password cracking session using several different input dictionaries. To do this with traditional methods, people often input each dictionary in their password cracker sequentially. They start by using a very small input dictionary of highly probable words, and apply many complicated word mangling rules to them. Later if that attack is not successful, they then run a different attack using a much larger input dictionary. This can best be seen in John the Ripper where it starts in "Single" mode using an input dictionary of user information gathered from the target system, and if that doesn't work, switches to "Wordlist" mode where a much larger input dictionary is used. The problem with this approach is that an attacker may want to try a low probability dictionary word with a high probability mangling rule, (such as 'zebra1'), before they try a high probability dictionary word with a low probability mangling rule, (such as 'p@sSword921'). That is very hard to do with traditional password crackers.

There are many other cases as well where it is hard to accurately represent how people create passwords using a rule based dictionary attack. For example, it becomes very difficult to model how multiple word mangling rules should be optimized when they are all applied to try and crack strong passwords. While the number '1' is much more likely to be used than the number '8', and the number '99' is more likely than the number '32', should the guess '1password32' be tried before the password guess '8password99'? How about 'pAssword1'? Should it be tried before the guess 'Password8'?. These are very tricky questions, and their answers can have a drastic impact on the effectiveness of a password cracking session.

What we're attempting to do is turn the problem on its head. Instead of trying to create word mangling rules that mimic the way people create passwords, we are instead assigning probabilities to user behavior, and then using those probabilities directly to generate very fine-grained word mangling rules, and from those, password guesses. It's an important distinction to make, and our testing so far has shown this approach to be very promising.

5.2 The Mechanics of Probabilistic Password Cracking

5.2.1 Using Probabilistic Grammars

Context-free grammars have long been used in the study of natural languages [38, 39, 40], where they are used to generate (or parse) strings with particular structures. As will be shown in the rest of this chapter, they also provide a very powerful framework to represent password guesses.

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):

$$\boldsymbol{\alpha} \rightarrow \boldsymbol{\beta} \tag{1}$$

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. As will be discussed in more detail in Chapter 5.2.2, we derive a set of productions and associated probabilities automatically from training sets of real life passwords. These productions attempt to capture the characteristics of how people create passwords, along with the frequency that certain word mangling rules are used. While we have experimented with many types of productions during the development of this algorithm, for simplicity this section only illustrate grammars that use the start symbol and 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* (ignoring case), *digit variables* and *special variables* respectively. Please note that rewriting of alpha variables is done using an input dictionary similar to that used in a traditional dictionary attack. Because of this, the full rewrite for alpha variables is not shown in many of the examples below due to the large size of input dictionaries traditionally used in a password cracking attack.

It should be noted that the *n* subscript in our variables, such as $\mathbf{D_n}$ is used to denote the length of the replacement variable. So a $\mathbf{D_2}$ represents a variable that would be rewritten as a two digit number. This means that the probabilities associated with the rewrite of a $\mathbf{D_1D_1}$ may be very different from those associated with a $\mathbf{D_2}$, even if they would produce the same guesses. This is due to the conditional probability associated with certain strings. For example, the two digit number '79', may have a very high probability due to it being someone's birth year, but the numbers '7' and '9' may have a very low probability of being selected independently. The key to avoid duplicate guesses then is to make sure no variables of the same type appear next to each other. For example, the RHS variable $\mathbf{D_1D_1}$

should never be allowed to appear in the grammar, but $\mathbf{D_1S_1D_1}$ is ok. This rule is automatically enforced during the training phase when we generate our grammar, and will be gone into more detail in Chapter 5.2.2

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 preprocessing phase, we also derive the probabilistic context-free grammar from the training set. An example of such a grammar, (along with an input dictionary containing two words), is shown in Table 5.2.1.1. Given this grammar, we can derive, for example, a terminal structure (consisting of all terminals):

$$\boldsymbol{S} \rightarrow \mathbf{L_3D_1S_1} \rightarrow \mathbf{L_34S_1} \rightarrow \mathbf{L_34!} \rightarrow \mathbf{cat4!} \quad (2)$$

with associated probability of 0.04875.

Table 5.2.1.1: Example probabilistic context-free grammar

LHS	RHS	Probability
S →	$D_1L_3\ S_2$	0.75
S →	$L_3D_1S_1$	0.25
D_1 →	4	0.60
D_1 →	5	0.20
D_1 →	6	0.20
S_1 →	!	0.65
S_1 →	%	0.30
S_1 →	#	0.05
S_2 →	$$	0.70
S_2 →	**	0.30
L_3 →	cat	0.50
L_3 →	hat	0.50

For optimization purposes, it is useful to introduce the notion of *containers*. A container is a structure that allows us to optimize computations related to terminals of similar type that all have identical probabilities. We use the term pseudo-terminal to indicate a container's representation in our grammar. Pseudo-terminals are an implementation convenience and need not actually appear in the formal grammar. For example, we currently assign all dictionary words of length *n* from the same input dictionary the identical probability $1/k_n$, where k_n is the number of words of length n in the input

dictionary. Thus we have an associated pseudo-terminal lower case L sub k (we call it $\mathbf{l_k}$) with an associated production:

$$\mathbf{L_k} \rightarrow \mathbf{l_k} \qquad \text{(with probability } 1/n_k\text{)} \tag{3}$$

This means the grammar views all n-length terminal words as being in the same container. In Table 5.2.2.1, $\mathbf{l_3}$ is a container. While not necessary for the theory of how our probabilistic password cracker works, in practice containers have a large impact in the efficient operation of the next function, which will be described in Chapter 5.2.4.

Just like the notion of containers, there are several stages from processing the start variable **S** to the final stage where all variables have been re-written as terminal value, where we have found it useful to define additional terms that allow us to better optimize our generation algorithms. These stages, along with example of a rewrite rule at those stages, can be seen in Table 5.2.1.2

Table 5.2.1.2: Listing of different grammar structures

Structure	Example
Simple	SLD
Base	$S_1L_8D_3$
Pre-Terminal	$\$L_8123$ or $\$l_8123$
Terminal (Guess)	$password123

The *Simple Structure* is the first re-write rule after the start symbol **S**, but it does not contain length information. While its use was investigated early on, it was quickly discovered that there is some very helpful context sensitive information that is lost when we used the simple structures in our grammar. For example, people generally create passwords that are between six and eight characters long. That information is not captured in the simple structure, and is hard to re-introduce when using a context-free grammar.

Because of that, we currently use the *Base Structure* as the first set of rewrite rules after the start symbol **S**. Base structures are exactly like simple structures, except that they contain length information for the following re-write rules. This way we can capture what essentially is context sensitive information while using a context-free grammar.

Sentential forms containing only pseudo-terminals or terminals are referred to as *pre-terminal* structures. These pre-terminal structures are very important because the probability associated with a pre-terminal structure no longer changes with any additional replacements, and thus is the same

probability as the final terminal guess. This allows us to work with pre-terminal structures when generating password guesses in probability order, instead of having to fill in all the variables with terminal values. Since a single pre-terminal structure can often generate thousands of unique terminal values, this is essential for having an efficient algorithm. This is also important for enabling distributed password cracking attacks. 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 individual container values, (often mostly just dictionary words, though this can also include digit and special strings), and hash the guesses. Having a single pre-terminal structure represent several thousand, (or more), password guesses reduces the need to send large amounts of data over the network. This ability to distribute work is a major requirement for any password cracking program to be competitive with existing alternatives. Note that we only need to store the probabilistic context-free grammar on each password cracking node, and the management server is the only node that needs to derive the pre-terminal structures.

Referring back to Table 5.2.2.1, an example of a pre-terminal structure that could be generated from that grammar would be $\mathbf{l_3 4!}$ with the associated probability of 0.04875. Note that $\mathbf{l_3}$ variable would point to the input dictionary containing three character long words, (in this case 'cat', and 'hat') and the final password guesses, (cat4!, hat4!), would also each have a final probability of 0.04875. It's important to remember that containers can also hold digits and special strings in addition to dictionary words. In the example grammar, the numbers '5' and '6' both fall into the same single digit container since their probabilities derived from $\mathbf{D_1}$ are the same. Let the associated pseudo-terminal be $\mathbf{d^{5,6}}$. Our algorithm would thus derive the pre-terminal structure $\mathbf{l_3 d^{5,6} !}$ with probability 0.01625. This pre-terminal structure would then be used to output the following terminal password guesses: {'cat4!', 'hat4!', 'cat5!', 'hat5!'} each with an associated probability of 0.01625.

It should be noted that the algorithms described in this chapter currently do not use the full capabilities of probabilistic context-free grammars since their production rules derived from the training sets do not have any recursion. Thus the production rules being used are equivalent to (probabilistic) finite state machines. However, modeling password creation strategies as probabilistic context-free grammars allowed us to find an efficient next function for generating the pre-terminal structures, as discussed in Chapter 5.2.4. Creating a next function would have likely been more difficult if a finite state machine model had been implemented. Furthermore, using context free grammars is a more general solution, as the *next function* described in Chapter 5.2.4 only requires that the probabilistic context-free grammar be unambiguous. Consider the rule:

$$\boldsymbol{S \rightarrow D_1 L_2 S L_2 S_1 \mid \#} \qquad \textbf{(4)}$$

where ***S*** is the start symbol. We don't learn such a rule at present from the data since it does not seem to be the way that passwords are created. Our next function however is still compatible with a grammar having this rule and the resulting language is not equivalent to any regular language. In future work, the more general capabilities of context-free grammars, in addition to their superior modeling capability, may prove to be particularly useful for attacking passphrases because of their obvious ability to represent phrase and language structures.

5.2.2 Basic Preprocessing

The preprocessing phase is where the actual probabilistic context free grammar is created. This is done by training our password cracker on a set of previously disclosed passwords. Once built, the resulting grammar can then be used in subsequent password cracking sessions. The advantage of this approach is that once a grammar is generated, it can then be distributed to other agencies without having to give them access to the original password lists the grammar was trained on. This allows a centralized agency to collect password lists to generate grammars without having to release those passwords lists to the general public.

Since the grammar is built during the preprocessing phase, all of the rules governing the grammar are determined in this phase. The first decision that needs to be made is how to decide what constitutes an alpha, digit, or special variable. 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 5.2.1.2 and Table 5.1.2.1. Please note that the character-set for alpha strings are defined by the training program and can be language dependent. For example, the Finnish character set contained the letters "äöïõ" along with normal ASCII alpha characters. In addition at this stage of the training, a distinction is not made between upper case and lower case letters.

Table 5.2.2.1: Listing of Different String Types

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

Now that we have defined alpha, digit and special strings, the next step is to capture their probability of occurrence. Each string of length *n* is counted, and a probability is assigned to each string. If no probability smoothing is applied, (which will be discussed in more detail in Chapter 5.2.4), a string *x*'s probability is assigned using the following formula:

$$\textbf{Probability}(x) = \frac{\mathbf{Xn}}{\mathbf{Kn}} \qquad \textbf{(5)}$$

where $\mathbf{x_n}$ is the number of occurrences of string *x* in the training set, and $\mathbf{k_n}$ is the number occurrences of strings of that type, (be it alpha, digit, or special), of length *n* that occurred in the training set. It's important to note that a string is defined by its greatest length of consecutive values from the same type. For example the digit string '79' would only be parsed as a D_2, and the '7' and '9' would not be parsed independently as two separate D_1 strings. The reason for this is that by only parsing strings of the same length, it allows us to save the conditional probability of the individual characters appearing together. In the above example the probability of '79' appearing might be high due to it being a birth year, even if the individual probabilities of '7' and '9' might be quite low. This can be seen in Tables 5.2.2.2 and 5.2.2.3 which show the probabilities of one and two digit numbers learned from the MySpace training set.

These strings and their associated probabilities are then stored in a text file as terminal replacements for their respective alpha, digit, and special variables of the same length. This way the grammar may be quickly generated for a password cracking session. For example all terminal replacements for the $\mathbf{D_2}$ variable are available with their associated probabilities in the digits2.txt file.

Table 5.2.2.2: 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 5.2.2.3: 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

The next step is to automatically derive all the observed base structures, along with their associated probabilities, from the passwords in the training set. The difference between base structures and strings though is that if no probability smoothing is applied, the probability assigned to each base structure is:

$$\textbf{Probability}(x) = \frac{\mathbf{Xn}}{\mathbf{Ktotal}} \quad \textbf{(6)}$$

where $\mathbf{x_n}$ is the number of occurrences of that particular base structure, and $\mathbf{K_{total}}$ is the total number of base structures parsed. This means that unless any password guesses are rejected from the training set, $\mathbf{K_{total}}$ equals the total number of passwords parsed. An example of that can be seen in Table 5.2.2.4 where the top ten base structures are displayed from the MySpace training set. As you can see, a vast majority of users choose a dictionary word for their password and then appended one or two digits to the end. This of course would change depending on the training set, as cultural backgrounds, age, password creation requirements, and importance of the password protected material. Therefore, several different grammars may be created based on different training sets, and in turn the appropriate grammar may then be selected based on which one is the closest match to the target's profile

Table 5.2.2.4: Probabilities of Top 10 Base Structures

Base Structure	Number of Occurrences	Percentage of Total
L_6D_1	2382	7.09752
L_6D_2	2181	6.49861
L_7D_1	2100	6.25726
L_8D_1	1746	5.20246
L_5D_2	1557	4.63931
L_9D_1	1366	4.07020
L_7D_2	1328	3.95697
L_5D_1	1283	3.82288
L_4D_2	1206	3.59345
L_8D_2	1152	3.43255

5.2.3 Case and Dictionary Preprocessing

The next step is to incorporate case mangling rules of alpha strings into our probabilistic grammar. This is required to target strong passwords where people use uppercase and lowercase letters when creating their password. While we originally looked at several different methods to do this, such as including case information into the base structures, we eventually settled on using case mangling rules as an additional replacement rule in our grammar. To do this end, we parse all alpha strings, regardless of their base structure, and extract a mask representing the capitalization rules used. We represent uppercase letters as **U**, and lowercase letters as **N**, (for **N**ot uppercase). For example, the alpha string, "PassWord" would be represented by a mask of $\mathbf{U_1N_3U_1N_4}$. Probabilities are then assigned to each mask based on the number of occurrences of each mask of length n, for all alpha strings of length n. See Table 5.2.3.1 for an example of this based on the MySpace training list.

As you can see from the results, if uppercase characters are not required, users very rarely incorporate them into their passwords. That being said, it still is useful to include case mangling information in our grammar even when there is no password creation policy since some password guesses with uppercase letters still are more probable than guesses generated with other mangling rules. This means it makes sense to use case mangling rules even during a regular cracking session.

TABLE 5.2.3.1: Probabilities of Top 5 Case Mangling Masks for Six Character Alpha Strings

Case Mask	Number of Occurrences	Percentage of Total
N_6	7080	93.206%
U_1N_5	241	3.1727%
U_6	222	2.9225%
N_3U_3	8	0.1053%
$U_1N_4U_1$	6	0.0078%

Once included in our grammar, the case mask is used as a replacement variable for alpha strings such as $\mathbf{L_6}$. The case masks themselves then are replaced with the terminal values from the input dictionary with the appropriate letters uppercased and lowercased. Because of this, the probability of the case mask is factored into the pre-terminal probability. In the example of the last section, a case mask of $\mathbf{UN_2}$, would then create the password guesses {'Cat4!', 'Hat4!', 'Cat5!', 'Hat5!'}. The important

thing to keep in mind that while **UN_2** may look like two separate variables due to the way we represent it, in actuality it is treated as one variable. Thus we can have the following rewrite rules:

$$UN_2 \rightarrow \textbf{Cat} \qquad (7)$$
$$UN_2 \rightarrow \textbf{Hat}$$

After including case mangling into our grammar, we then need to assign probabilities to specific alpha string replacements to model the probability that common base-words (such as football, password, and iloveyou) are used. Rather than assigning probabilities to the words found in the training set, probabilities are instead assigned to input dictionaries that are chosen by the attacker. This is because we feel that the alpha strings found in the training set represent only a fraction of the alpha strings likely to appear in the target set of passwords. By making use of commonly used input dictionaries, we can gain a larger coverage of the target set, while the use of multiple dictionaries allows us to assign higher probabilities to more likely words. For example, the attacker can create a very small dictionary of common words, and then a much larger dictionary of less probable words to use in their password cracking attacks.

To assign probabilities to words in multiple dictionaries, we use a modified form of edit distance to see the percentage of passwords that are likely to be cracked in the training set by each input dictionary. A more detailed discussion on how edit distance is applied can be found in Chapter 3.3. For example, suppose the attacker chooses to use two different input dictionaries and according to our edit distance calculations the first dictionary would crack 60% of the passwords in the training set while the second dictionary would crack 5% of the training set. The initial probabilities that each dictionary word would receive would be:

$$(1/n_k) * \textbf{(Percentage of passwords cracked)} \qquad (8)$$

where n_k is the number of words in the input dictionary of length k. Therefore if the first dictionary contained 10,000 words of length six, each six letter word would be assigned a value of 0.006%. If the second dictionary only contained 500 words of length six, then each of them would then be assigned a value of 0.01%. This means that even though the second dictionary cracked fewer passwords, due to its small size each word is given a significantly higher probability. After these percentages are calculated, normalization is applied so that all of the dictionary word probabilities across every single dictionary used adds up to 100%. This means if only one input dictionary is used, the probability assigned to each word would simply be $(1/n_k)$. Additional logic is applied in our training program to deal

with word duplication, (the same word appearing in multiple input dictionaries), by assigning duplicate words to the dictionary which would grant it the highest probability.

The information described above is also captured automatically from an input file of training passwords using our preprocessing program. The training process itself generally takes less than ten minutes, (a vast majority of this is due to calculating the probabilities of individual dictionaries), though it varies depending on the size of the training set and the number and size of input dictionaries used.

5.2.4 Probability Smoothing

As can be expected, the training set clearly does not cover all of the possible values that might be encountered in a password cracking attack. For example, someone could create the password, 'monkey762', but the number '762' may have never been present in the training set. Ideally, an attacker would like to try these values, but at a reduced probability compared to what was found in the training set. To that end, our grammar uses probability smoothing to assign probabilities to values not found in the training set. The current method is to use a variant of Laplacian smoothing where the probability of an element is assigned by the following formula:

$$\frac{\text{(Number of Times Seen in the Training Set)} + \alpha}{\text{(Total Number of Values)} + (\text{Number of Categories} * \alpha)} \quad (9)$$

In this equation **α** represents a value between 0 and 1, where 0 represents no probability smoothing and 1 represents a large degree of probability smoothing. This equation is then applied to all digit strings, special strings, case masks and base structures. Therefore, for an **α** value of 1, if 500 two digit numbers were observed in the training set, but the number 33 was never encountered, the number 33 would be assigned a probability of:

$$\frac{(0) + 1}{(500) + (100 * 1)} \approx 0.0016 = 0.16\% \quad (10)$$

As can be seen, this is considerably less probable than the 1.0% it would be assigned if the probabilities were assigned uniformly across all two digit numbers.

Probability smoothing may also be applied to base structures in much the same way. While theoretically the number of base structures is unbounded, since someone could always create a password one character longer than their previous one, in reality the length of passwords targeted by

an attacker is finite. Therefore, it is up to the attacker to decide the maximum length of base structure to apply probability smoothing to. For the test detailed in this paper, probability smoothing was only applied to base structures up to six characters long. Therefore, given enough time, the resulting grammar eventually brute forces all base structures up to six characters long. It should be noted then that this is the equivalent of a targeted brute force attack. Currently alpha strings do not have probability smoothing applied to them. Once that is implemented though, this grammar would generate rules such as “brute force all six character long words, and append the number ‘123’ to the end”. Ideally we would like to also implement Markov frequencies into alpha string probability smoothing. This could be incorporated into our current grammar by creating sub-dictionaries of different probability brute force guesses.

5.2.5 Effectively Generating a Next function

Constructing a probabilistic context free grammar is only the first step. For it to be useful in a password cracking session, we also need an algorithm that takes that grammar as input and generates password guesses in probability order. Detailed in this section is one such algorithm to do that. First note that it is trivial to generate the most probable password guess. One simply replaces all the base structures with their highest probability terminals and then selects the pre-terminal structure with the highest probability For example, using the data in Table 5.2.2.1, the highest probable pre-terminal structure would be **$4L_3\$\$$.** Since the number of base structures is fairly low, (there are only 1589 base structures generated by the MySpace training set before probability smoothing), 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. 5.2.5.1, we would like to generate the pre-terminal structures ‘{5,6}{cat,hat}$$’ and ‘4{cat,hat}%%’ (nodes 2 and 3) only after ‘4{cat,hat}$$’ (node 1) has been generated.

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. This is the method that was described in some of our earlier work [48]. Unfortunately this pre-computation step is not parallelizable with the password cracking step that follows (i.e., it is not an online algorithm). It also resulted in over a

hundred gigabytes of data even for limited runs 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.

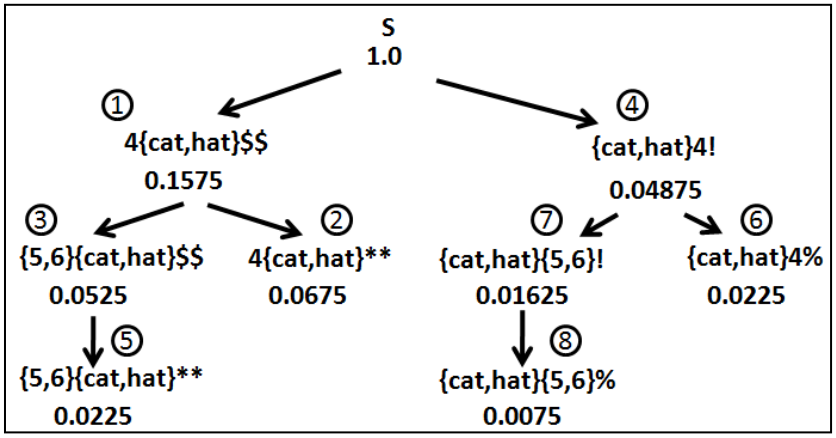


Figure 5.2.5.1 Parse Tree for the Pre-terminal Structures for the Base Structures in Table 5.2.2.1

Another tactic would be to take the method detailed in the Narayanan paper [19], and only generate the pre-terminal structures whose probability is above a set limit. This is also the approach used in John the Ripper's Markov mode. The problem with that algorithm is that it would not generate password guesses in true probability order. As seen in Chapter 2.1.3, depending on the limit set, all that algorithm does is guarantee that the probability of all password guesses generated falls above a certain limit, but the actual order those guesses are generated in is fairly random.

Yet one more approach would be to use a typical depth-first search algorithm, but unfortunately as every node needs to be visited eventually, the size of our final tree can include several trillion nodes. Since performance is important, these algorithms did not prove feasible due to the amount of backtracking required.

The current 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 $\mathbf{L_3D_1S_1}$ the variable $\mathbf{L_3}$ would be assigned an index of 0, $\mathbf{D_1}$ an index of 1, and $\mathbf{S_1}$ an index of 2. Next, all terminal values, (such as the numbers 4, and 5 for $\mathbf{D_1}$), are sorted in priority order for their respective class. That way the next most probable terminal value can be quickly found.

The structure of entries in the initial priority queue can be seen in Table 5.2.5.1. Each entry contains 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. In short, it makes sure that only one unambiguous parse tree is generated from all of the different parse trees that are possible.

Table 5.2.5.1: Initial Priority Queue for the Grammar in Table 5.2.1.1

Base Structure	Pre-Terminal	Probability	Pivot Value
$D_1L_3S_2$	4{cat,hat}$$	0.1575	0
$L_3D_1S_1$	{cat,hat}4!	0.04875	0

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 5.2.1.1. Initially all the highest probability pre-terminals from every base structure is inserted into the priority queue with a pivot value of 0. See Figure 5.2.5.1 and Table 5.2.5.1

Next, the top entry in the priority queue is popped. The pivot value then is consulted, and child pre-terminal structures are 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. 5.2.5.1 and Table 5.2.5.2 to view the result after popping the top queue entry. Also see Appendix 1 for a pseudo-code implementation of the full algorithm.

Table 5.2.5.2: Priority Queue after the First Entry is Popped

Base Structure	Pre-Terminal	Probability	Pivot Value
$D_1L_3S_2$	4{cat,hat}**	0.1575	2
$D_1L_3S_2$	{5,6}{cat,hat}$$	0.1575	0
$L_3D_1S_1$	{cat,hat}4!	0.04875	0

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 additional containers to fill in for it. This is due to the fact that there was only one input dictionary, so the algorithm views all words of length n from that dictionary as a single container. Both the D_1 structure and S_2 structure were replaced, resulting in two new pre-terminal structures being inserted into the queue with pivot values of 0, and 2. Notice that when the next priority queue entry is popped, it does not cause a new entry to be inserted into the priority queue since it can only increment the S_2 structure due to the pivot value being equal to 2. As for the S_2 structure, since '**' is the least probable terminal variable, there is no next-highest replacement rule and this entry is simply 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. Also, it's important to remember that each pre-terminal structure often generates many different terminal values, (password guesses), due to the use of containers. This means each node popped frequently produces several thousand guesses.

5.2.6 Proof of Correctness

First, we establish a few claims about how the algorithm introduces pre-terminal structures in the priority queue for processing.

Claim 1. Every pre-terminal structure generated by the grammar eventually is added to the priority queue for processing.

To prove this claim, we only have to show that it holds for pre-terminal structures with maximum probability among all pre-terminal structures sharing the same base structure. Indeed, let X_i be an arbitrary pre-terminal structure and assume that the maximum probability pre-terminal structure X_{max}

that shares the same base structure as X_i is added to the priority queue at some point. Consider the first position where X_{max} and X_i differ. The probability of the pre-terminal in that position is higher at X_{max} than at X_i, otherwise, by changing the pre-terminal at X_{max} we would obtain another pre-terminal structure with the same base structure as X_{max} but higher probability, contradicting the definition of X_{max}. So, by processing X_{max} at this position as pivot will eventually lead to X_{max}' which differs from X_{max} in a single position, where it matches X_i. The argument can be repeated for subsequent positions of the pivot, to show that eventually X_{max} will produce X_i.

The claim now follows because the algorithm initialization is defined precisely by introducing the highest pre-terminal structure for each base structure to the queue.

Note that since our grammar does not include empty productions (i.e., a variable with a production leading to the empty string in the RHS), the processing of two distinct base structures cannot lead to the same pre-terminal structures. Moreover, since each pre-terminal structure is processed by the algorithm by replacing at each pivot position with another entry of lesser probability value, it is clear that Claim 1 has a complement:

Claim 2. No pre-terminal structure generated by the grammar is added to the priority queue for processing more than once.

Having established these preliminary statements, we are now ready to prove the main result:

Property 1. Pre-terminal structures are output in non-increasing probability order.

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 *Property 1* 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. **(11)**

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 (recall via *Claim 2* that an element is added to the queue at most once), 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 *Property 1*. Consequently, when *z* was output, it did not violate this property, and since:

$$\mathbf{Prob(z) >= Prob(x) > Prob(y)} \qquad \mathbf{(12)}$$

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

5.2.7 The Deadbeat Dad Algorithm

When talking about the *next algorithm*, it is important to remember that it actually performs two different tasks. The first is that it provides a unique and repeatable parse tree given a probabilistic context free grammar. What this means is for every single terminal value that exists in the grammar, there is exactly one set of productions that generate it. The next algorithm accomplishes this through the use of a pivot value. What the pivot value does is ensure for every child node in the parse tree, there is exactly one parent that is allowed to produce it. Consider the parse tree built by the next algorithm and shown in Fig. 5.2.7.1. Instead of using the password cracking grammar described previously, this parse tree simply shows a generic context free grammar containing one base structure with three different non-terminal variables. For each variable, there are then three possible terminal replacements. Therefore each node in the tree represents one final terminal value generated by this grammar.

The second task that the next algorithm performs is given a unique parse tree; it generates all possible strings in probability order. This is primarily accomplished through the use of a priority queue. As we will see in chapter 5.3.3 though, while the next algorithm has performed very admirably in all password cracking sessions to date; due to its reliance on storing values in the priority queue, the next function's memory requirements have the potential to become a bottleneck when more complicated grammars are introduced. To reduce the memory usage caused by storing a large number of nodes in the priority queue, a new "next" algorithm needed to be developed. The replacement version for the

next function currently is labeled the deadbeat dad algorithm. While the algorithm's name is unfortunate, it has stuck because it fairly accurately describes how the algorithm works.

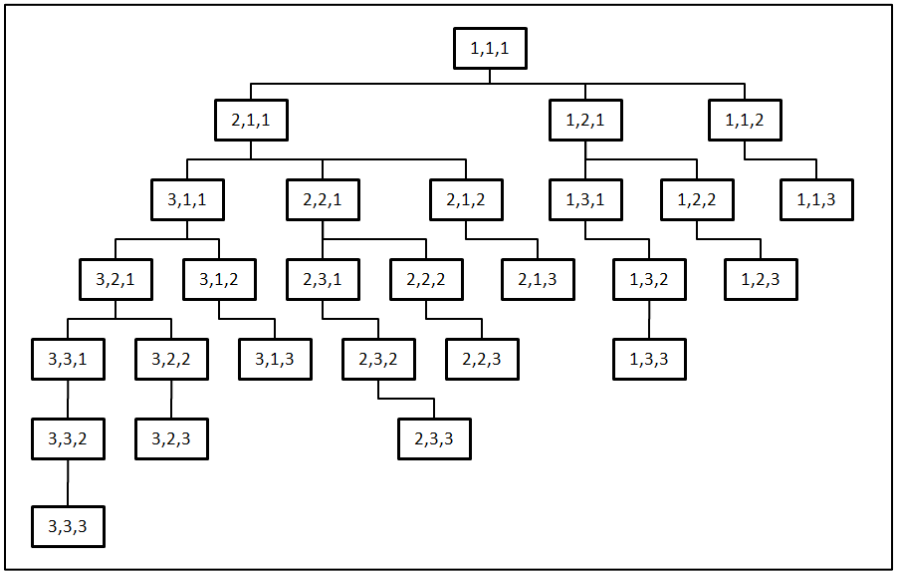


Figure 5.2.7.1: Generic Parse Tree Generated by the Next Function

The key to reduce the memory usage is to delay pushing child nodes into the priority queue as long as possible. With the original next algorithm, when a parent node is popped from the priority queue, all of the parent node's possible children as determined by the pivot value are then pushed into the priority queue. The goal of the deadbeat dad algorithm then is to have some parent nodes after they are popped occasionally abandon their child nodes and not push them into the priority queue. This is possible due to the fact that if a pivot value is not used, there often exists many different sets of productions that can generate the same final terminal value. This can be seen in Fig 5.2.7.2 which shows the possible productions for the node {1.2.2}, using the simplified grammar depicted in Fig 5.2.7.1. The probability associated with each terminal value is listed to the right of each node.

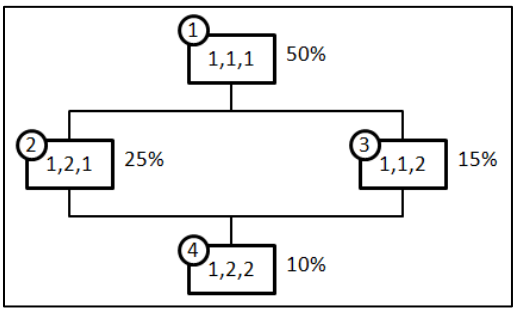


Figure 5.2.7.2: Multiple Parents for Node #4

The current next function first pops node 1, and then pushes nodes 2 and 3 into the priority queue. Node 2 then is popped, due to its probability being 25%, and the function then pushes node 4 into the priority queue. This is because node 4 is node 2's child as seen in Fig. 5.2.7.1. The problem is a child node's probability is always less than any of its parent's probability, as shown in *property 1* in the proof of correctness detailed in Chapter 5.2.6. This means node 4 will never be popped from the priority queue until node 3 is popped. Therefore node 3, instead of node 2, should be responsible for pushing node 4 into the priority queue.

Since no child nodes are popped until all of their parent nodes are popped, and the parent nodes are popped from the priority queue in probability order, ideally a child node should be inserted into the priority queue by the parent node with the lowest probability. This in a nutshell is the deadbeat dad algorithm, as can be seen in Figure 5.2.7.3.

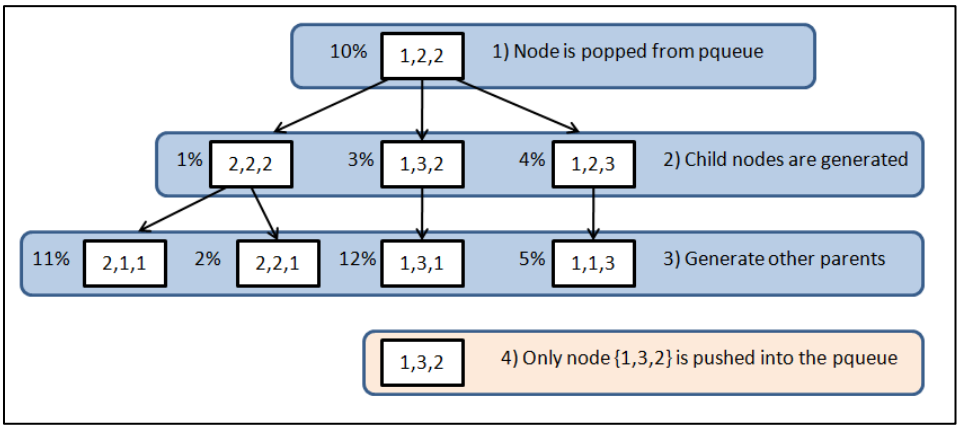


Figure 5.2.7.3 What Happens When a Node is Popped in the Deadbeat Dad Algorithm

The pseudo code for this algorithm can also be seen in Appendix B. This algorithm uses the same priority queue as the next function. The difference is, whenever a node is popped, it then creates all of the possible children for that node. For each child generated, the algorithm then generates all of the possible parents for that child and check to see if any of them have a lower probability then the currently popped parent. If another parent's probability is less, the popped node abandons the child for the other parent to take care of. If another parent node's probability is the same as the popped node's probability, then the tiebreaker is assigned by whichever node created the child with the rightmost transition. This way, one parent is guaranteed to take care of the child. If the popped node is the lowest probability parent, it then pushes the child node into the priority queue.

The advantage of this approach is that it results in a significantly smaller priority queue. Even with the additional overhead of having to calculate the probability of possible parents for each child

node, the smaller queue size leads to faster insertions, which in turn reduces the algorithm's running time along with its memory usage. More analysis of the performance characteristics of the deadbeat dad algorithm can be found in Chapter 5.3.3

5.3 Experiments and Results

The current version of our probabilistic password cracker, called 'UnLock', is written in a combination of Java and C++, and is available for free download from [49]. A screen shot of our program can be seen in Figure 5.3.1.1 showing our password cracker being trained on a known password set.

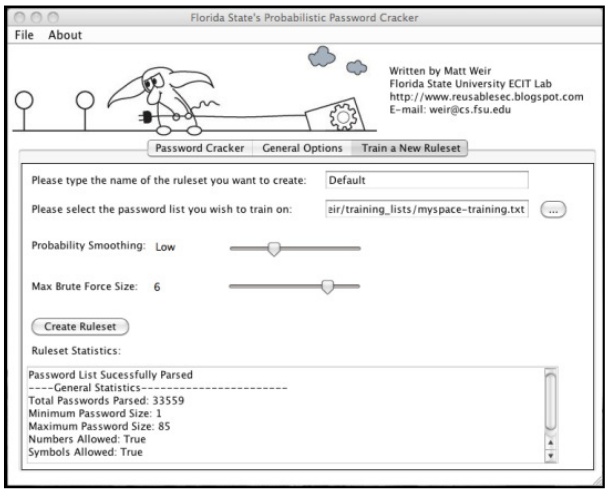


Figure 5.3.1.1: Screen Shot of our UnLock Probabilistic Password Cracking Program

As stated previously, our password cracker is trained on existing corpuses of disclosed passwords to create a probabilistic grammar. It then uses this grammar to generate password guesses to crack passwords. If the target passwords are not hashed, UnLock simply outputs its guesses to a checker program that verifies if the guesses match a valid password. If the target password is hashed, we instead hash the guess using the appropriate hashing function before checking to see if the generated hash matches one or more of the target hashes. The password is considered crack it the two hashes match. For all of the tests that follow, if we included a user's password in our training set for a particular experiment, we do not use that same user's password in the test set for that attack. This is just another way of saying that we do not train and test against the same passwords.

As a comparison against our probabilistic password cracking technique, it was 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.

5.3.2 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. A total of six publicly available input dictionaries were selected to use in the tests below. Three of them, “Passwords.lst”, “Spanish_lower”, and “Common_Passwords” were obtained from John the Ripper’s public web site [23], with “Passwords.lst” being the default dictionary distributed with John the Ripper. Additionally we used the input dictionary “dic-0294” which we obtained from a popular password-cracking site [50]. This list was chosen due to the fact that it has been found to be very effective in the past when used in traditional password crackers. The dictionary “Wordlist.txt” which is the default input dictionary for the popular password cracker Cain&Able [24] was also included. Finally, I created my 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 5.3.2.1. 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 in Table. 5.3.2.1. Because our probabilistic password cracker can make use of multiple input dictionaries, in some of the tests in section 4.3, both Dic-0294, and the Common-Passwords input dictionaries were used. To keep the comparison fair with John the Ripper, Dic-0294 actually contains all of the words found in the Common-Passwords input dictionary. This means all this optimization is doing is giving several of the words in Dic-0294 a higher probability compared to the rest. Also for these test, UnLock is still limited to the number of guesses John the Ripper makes just using Dic-0294.

Table 5.3.2.1: Size of Input Dictionaries

Dictionary Name	Number of Dictionary Words	Number of Password Guesses
Dic-0294	869,228	37,781,538
Passwords.lst	3,115	139,498
Common_Passwords	816	41,019
English_Wiki	68,611	3,179,219
Wordlist.txt	306,706	15,017,243
Spanish_Lower	86,060	3,405,215

5.3.2 Password Cracking Results

Since our probabilistic method requires it to be trained it on real passwords, for every test run there is a training set and a test set. For the first test the MySpace test set was targeted while training UnLock on the MySpace training set. Referring back to Chapter 1.6.1, the MySpace training set contained a total of 33561 passwords, and the MySpace test set contained a total of 33481 passwords. The results of our password cracker, UnLock, versus John the Ripper using different input dictionaries can be seen in Figure 5.3.2.1.

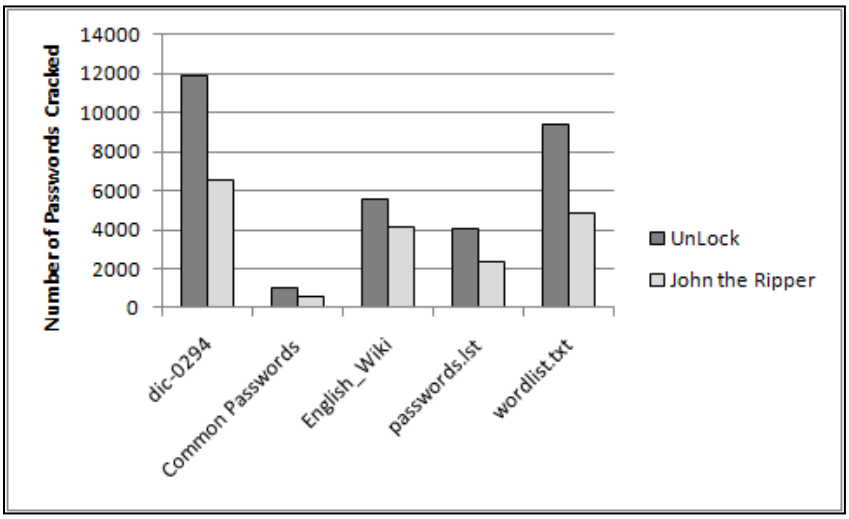


Figure 5.3.2.1: Number of Passwords Cracked Total. Trained on the MySpace Training List. Tested on the MySpace Test List

Given the same number of guesses, our probabilistic password cracker was able to crack between 36% and 93% more passwords than John the Ripper. For example, using the dictionary dic-0294, and given close to 38 million guesses, UnLock cracked 35.4% of the MySpace test set, while John the Ripper's default rules only cracked 19.6% of the test set.

While Fig. 5.3.2.1 shows the final password cracking results, it also is helpful to see how many passwords are cracked given a certain number of guesses. To that end, instead of showing only the final passwords cracked it is useful to see the number of password cracked over time using one input dictionary. The results shown running the above test using Dic-0294 can be seen in Fig. 5.3.2.2.

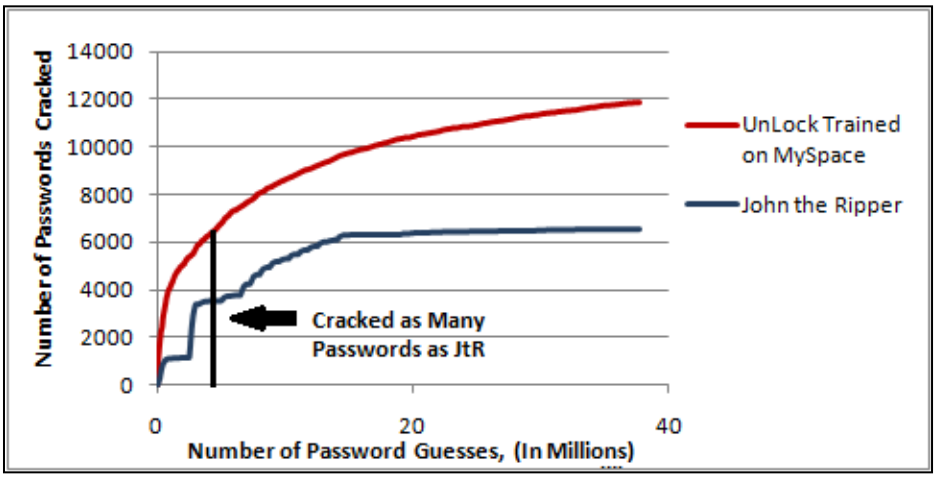


Figure 5.3.2.2: Number of Passwords Cracked Over Time. Trained on the MySpace Training List. Tested on the MySpace Test List

What stands out is that UnLock's graph is much smoother than John the Ripper's. This is an advantage of using a probability model since it can create very fine-tuned guesses compared to what is normal for a typical rule based mangling approach. Also, while the number of new passwords cracked goes down over time as it tries less and less probable guesses, you may notice that even after 37 million guesses, the number of new passwords that UnLock is cracking is much higher than John the Ripper. This leads to our next experiment. What happens when a password cracking session is run for a much longer period of time?

To model a longer password cracking session using John the Ripper, we ran John the Ripper in wordlist mode using dic-0294 just as we did in the previous test. To generate additional guesses, after the wordlist mode finished, we then ran John the Ripper using incremental mode to perform a brute force attack. This simulates a standard password cracking session as an attacker first tries a dictionary attack and then move on to brute force if that attack fails [51]. It's important to note that John the Ripper's incremental mode is actually quite intelligent, and uses trigraphs to model the conditional

probabilities of letters appearing together, (aka Markov models), as described in Chapter 2.1.3. A cracking session showing UnLock, John the Ripper using wordlist and incremental mode, and John the Ripper just using incremental mode can be seen in Figure 5.3.2.3.

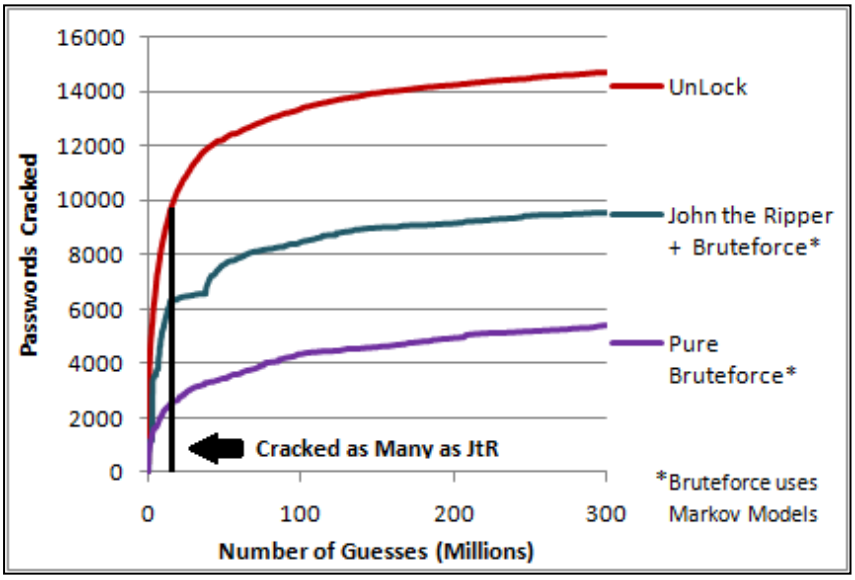


Figure 5.3.2.3: Number of Passwords Cracked Over Time. Trained on the MySpace Training List. Tested on the MySpace Test List

One interesting point is that when John the Ripper switched to a brute force attack, it almost immedialty cracked a large number of passwords. This was due to the number of shorter passwords that were vulnerable to a brute force attack but were not created using dictionary words that appeared in dic-0294. With probability smoothing, UnLock inhearently does a limited amount of brute force attacks as well, though since we currently do not apply probability smoothing to alpha characters, it does not try words not found in the input dictionary. This means that it may currently be worthwhile to pause our password cracking session and try bruteforcing a small keyspace before resuming UnLock. That being said, even after 300 million guesses, our probabilistic password cracker was still cracking more new passwords than John the Ripper.

So how does UnLock fair against other password lists? The next test shows UnLock, trained on the MySpace training set, and John the Ripper cracking the PhpBB list. As described in Chapter 1.6.2, this list contained 259,424 unsalted MD5 password hashes. Since all the passwords were hashed, (aka we don't know what they are until we crack them), all of the password hashes were used in the test set. Fig. 5.3.2.4 shows the results.

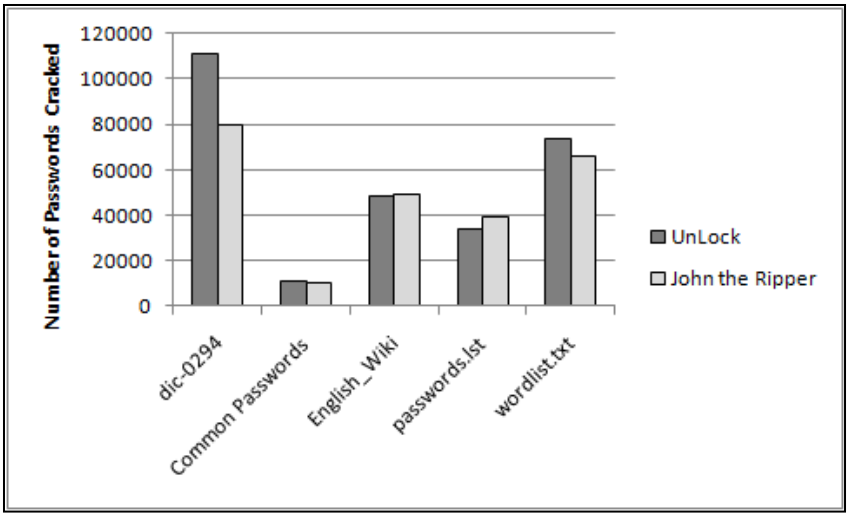


Figure 5.3.2.4: Number of Passwords Cracked Total. Trained on the MySpace Training List. Tested on the Phpbb.com Test List

Compared to the previous test shown in Fig. 5.3.2.1, the results are not nearly so dramatic, with our probabilistic method actually performing worse than John the Ripper in two of the cases. The main reason for this was the relative simplicity of the phpbb.com passwords. Using a much longer cracking session and multiple techniques, we have since been able to crack over 97% of the passwords in that set [18]. Analyzing those passwords, only 3% of the cracked passwords contained an uppercase letter, only 1% of the passwords contained a special character, and slightly more than 50% of the passwords only contained lowercase letters. In comparison, with the MySpace test list, 7% of the passwords contained an uppercase letter, 9% contained a special character, and less than 8% of the passwords contained only lowercase letters. This means that UnLock spent much more time trying complex password guesses when it would have better to only try basic password guesses. That being said, with a longer cracking session, like that used with dic-0294, UnLock still managed to crack 40% more passwords that John the Ripper's default rules.

The next test involves the Hotmail-10k list. As mentioned in Chapter 1.6.3, this list was comprised primarily of Portuguese and Spanish speakers [5]. Since the list contained plaintext passwords, it was possible to divide it up into training and test lists, each containing 4872 passwords.. The results shown in Fig 5.3.2.5 represent three different password cracking sessions using the specified input dictionaries. The first one is John the Ripper. The second session is UnLock trained on the MySpace training list. The third session is UnLock trained on the , and the Hotmail training list.

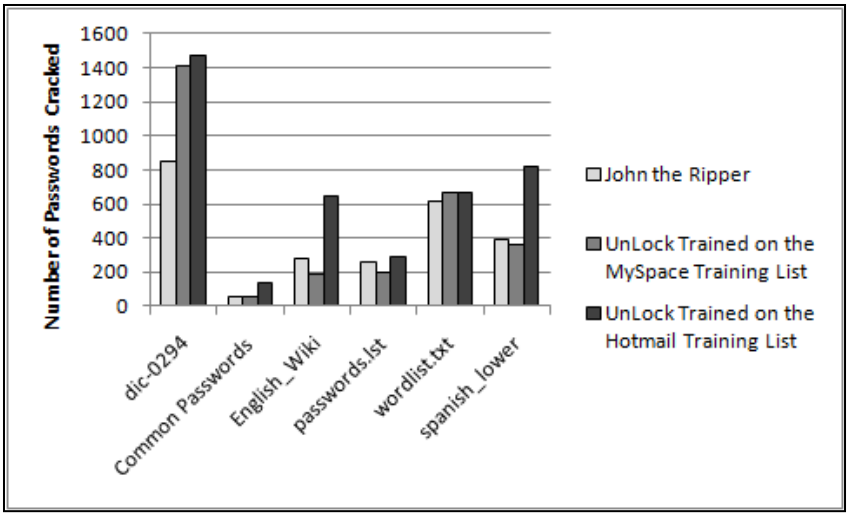


Fig. 5.3.2.5: Number of Passwords Cracked Total. Tested on the Hotmail Test List

The results of this test are pretty much as expected with our probablistic password cracker performing best when it was trained on Spanish and Portuguese passwords, (aka the Hotmail training list). The improvements when using a Spanish input dictionary were particularily striking. When the Spanish input dictionary was used and UnLock was trained on English passwords it performed slightly worse than John the Ripper, but when it was trained on the Hotmail set it cracked 107% more passwords than John the Ripper. That being said, when a longer password cracking session was run, such as when dic-0294 was used as the input dictionary, even UnLock trained on English passwords performed significantly better than John the Ripper.

5.3.3 Time and Memory Requirements

For any password guess generation algorithm, it's important to understand the computational time required to generate guesses. Furthermore, since our Next function is heavily reliant on a priority queue that grows over time, we also need to understand the memory requirements for our algorithm as well. First though, let's discuss the space complexity of storing the grammar which would be distributed to the end user once the password cracker has been trained.

Since the grammar is generated from a 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 Table 5.2.2.1. 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) depends 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. Table 5.3.3.1 details the size of several grammars created using an earlier version of our program. The two lists MySpace10k, and MySpace20k represent sub-lists of the MySpace training set respectively containing 10 thousand and 20 thousand training passwords.

Table 5.3.3.1: Size of the Stored Grammar Training Set

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

As you can see, the resulting grammar is extremely small. The number of guesses it can generate though is non-trivial taking into account 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 $S_1L_8D_3$. A pre-terminal value might take the form $\$L_8123$, 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 $S_1L_8D_3$ 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 5.3.3.2 for the total search space generated by each given training set and input dictionary using an older version on UnLock that did not include case mangling or probability smoothing. Please keep in

mind that newer versions of UnLock that include probability smoothing and case mangling generate substantially more pre-terminal and terminal structures than what is listed in Table 5.3.3.2. As should be apparent though, even with a very small input dictionary such as Common_Passwords, the number of possible guesses that can be generated by any of these grammars is much larger than any password cracking session is likely to generate. Therefore in just about any use case, UnLock runs until all the target passwords are cracked or the attacker decides to terminate the cracking session due to time constraints.

Table5.3.3.2: 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

The running time for our next algorithm for generating guesses is extremely competitive with existing password cracking techniques. As we add new grammar productions though, such as case mangling and probability smoothing, it has been increasing the computational time required to generate additional password guesses. On one of our lab computers, (MaxOSX 2.2GHz Intel Core 2 Duo) it took on average 33 seconds to generate 37,781,538 un-hashed guesses using an older version of our method that did not include case mangling or probability smoothing. Comparatively, the popular password cracking tool John the Ripper [23] 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.

When case mangling was included, (but not probability smoothing), the average time to compute 37,781,538 password guesses increased to 1 minute and 40 seconds, and the time it took to generate 300 million guesses increased to 19 minutes 41 seconds. Likewise when we included probability smoothing as well, the average time to compute 37,781,538 guesses increased to 4 minutes

and 53 seconds, and the time it took to generate 300 million guesses was 18 minutes fifty seconds. A majority of this time increase was probably due to the size of the priority queue increasing due to the more complex replacements used. Another possible option though could simply be that the case mangling rules were coded inefficiently. 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 several minutes or even hours spent generating guesses would be minor.

The next question is how does the space requirements of the priority queue grow over time, especially when new grammar productions are added. The priority queue grows when a popped parent node creates more than one child nodes. The size of the priority queue shrinks when a popped parent node has no children. By adding probability smoothing, the probability of a childless parent node being popped is often very low because some of the child productions usually include values not seen in the training set. The resulting pre-terminal values thus generally have a significantly lower probability then most pre-terminal values that only include values seen in the training set. These children then are very unlikely to be popped unless a much longer password cracking session is run. Likewise, when case mangling is added, there are some case masks, such as $U_1N_1U_1N_2U_1$ that can also have a very low probability. Once again this means that childless nodes generally include these low probability case mangling rules and therefore are unlikely to be popped from the priority queue during a normal cracking session. For an example of this, the priority queue for UnLock using the dictionary Dic-0294 and running without case mangling or probability smoothing enabled can be seen in Fig 5.3.3.1.

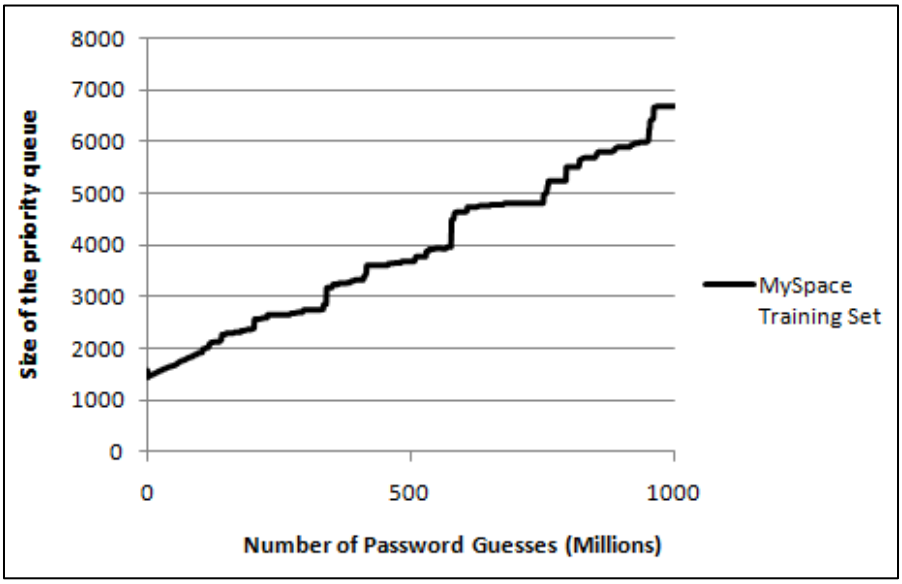


Figure 5.3.3.1: Size of Priority Queue

For comparison, Fig 5.3.3.2 shows the size of the priority queue when the Next algorithm is run two more times. Once when case mangling is enabled, but probability smoothing is not, and once when both probability smoothing and case mangling are enabled.

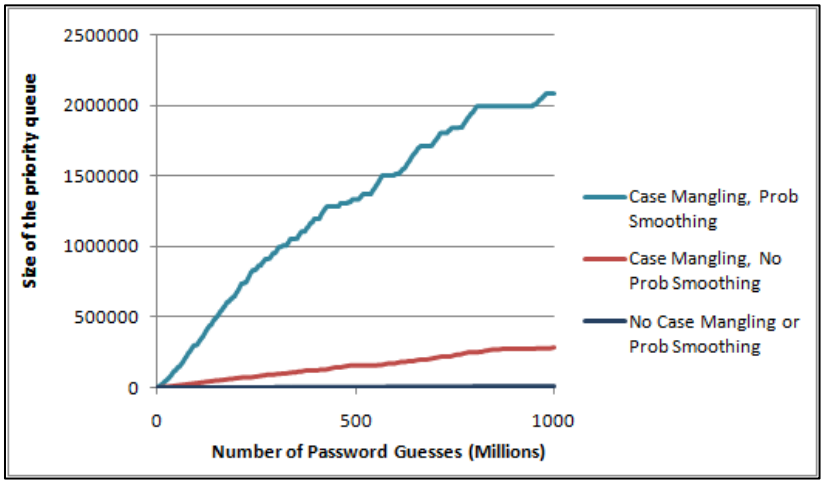


Figure 5.3.3.2: Production Rules vs. Size of the Priority Queue

As you can see, while case mangling adds significantly to the size of the priority queue, the queue size doesn't explode until after probability smoothing is added. Currently for most cracking sessions run, even with probability smoothing enabled, the size of the queue is still manageable but it was a point of worry. To enable more complicated grammars, the deadbeat dad algorithm as described in Chapter 5.2.7 was developed. When run using the grammar generated from the MySpace training set with case mangling and probability smoothing enabled, its priority queue grew at a noticeably slower rate than the original next function. The results of that test can be seen in Fig. 5.3.3.3.

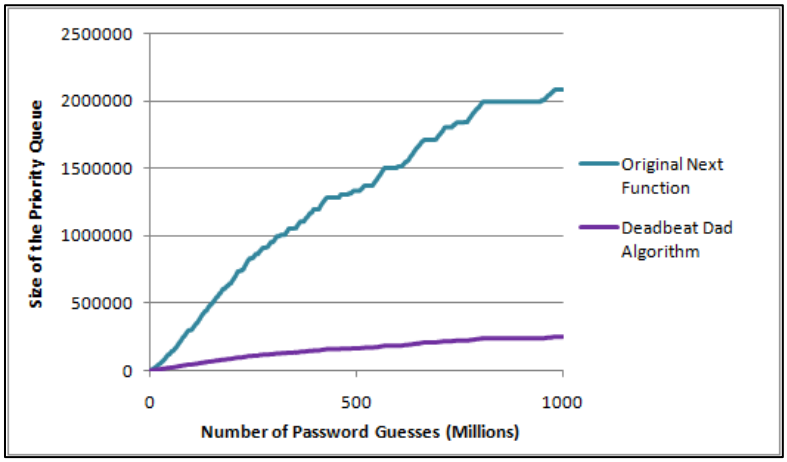


Figure 5.3.3.3: Size of Priority Queue - Deadbeat Dad vs. Original Next Function

As can be seen from the graph, the deadbeat dad algorithm drastically reduced the size of the priority queue. In fact when comparing the size of the priority queue generated by the deadbeat dad algorithm using a grammar that included probability smoothing to the original next function using a simpler grammar with no probability smoothing enabled, (as seen in Fig. 5.3.3.2), the deadbeat dad algorithm still performed slightly better. That's encouraging since this means that with the deadbeat dad algorithm, further productions may be added to the underlying grammar without having to be too concerned about the memory requirements. In the future though, if the size of the priority queue does become an issue there still are several other changes that may be made. The easiest would be to add a probability limit, below which child nodes would no longer be inserted into the priority queue. Since these low probability pre-terminal values would be unlikely to ever be popped from the priority queue during a real password cracking session, this approach would have minimal impact on the algorithm's effectiveness. If setting the correct limit is a concern, (such that an attacker may want to continue a password cracking session past the original limit they set), the pre-terminal values can be saved unsorted to disk. That way when the current priority queue is exhausted, a new limit may be set and all of the pre-terminal values saved to disk that have a probability above the existing limit can be re-inserted into the priority queue.

5.3.4 Future Research

There are several areas that are open for improvement with regards using probabilistic grammars for password cracking. First, UnLock currently does not support the idea of attacking keyboard combinations. Keyboard combinations include passwords that are memorable based on the location of keys on the keyboard vs. an actual word. An example of this would be the password 'qwerty' or '1qaz@WSX'. One possible solution would be to identify keyboard combination in the training phase. An attacker could then create a specialized input dictionary containing only keyboard combos and treat them like alpha strings, or they could create a whole new base structure, labeled K, that would represent keyboard combinations. One problem with these approaches is that they might generate duplicate password guesses. For example, the password guess '1qaz' could be made first by a keyboard combo, and again by the structure D_1L_3.

Another improvement would be to identify and attack letter replacement strategies. For example, changing the letter 'a' to an '@' in the password 'p@ssword'. One possible strategy would be to identify and assign a probability to specific replacements during the training phase through the use of edit distance. An attacker could then apply letter replacements using the specified probability, much like we currently apply capitalization mangling now. The main downside is that once again this may lead to

duplicate password guesses. For example, the password guess 'c@t' could be created both by a letter replacement rule and the structure $L_1S_1L_1$.

In addition, it would be nice to add support for tailoring a grammar against a particular target. Currently UnLock is optimized for cracking a large set of passwords. In an individual password cracking session though, the probabilities assigned to certain replacements should be modified to best mimic how the targeted individual created their password. One way this can be accomplished is by collecting information about the subject, such as birthdays, home addresses, names of family members, etc, and then creating specialized input dictionaries with associated high probabilities. It is also possible to modify certain digit and special string probabilities, such as dates, such as lowering the probabilities of dates that do not have a special relationship with the target while raising the probability of dates that do.

Yet another improvement, briefly alluded to in Chapter 2.1.4, is to add full targeted brute force support to UnLock. Currently UnLock do not perform probability smoothing on alpha strings. This means if an alpha string does not appear in one of its input dictionaries, it never tries that word. Ideally UnLock would like to use a form of Markov modeling, so that it would automatically switch between dictionary based attacks and brute force depending on the current probability of the password guess.

Finally, the next step is to expand the probabilistic method to attacking passphrases as well as passwords. Since probabilistic context free grammars are often used in parsing human generated text, the methods developed for attacking passwords may also be well suited for generating passphrase guesses. The main problem slowing this research is a lack of passphrase sets to train on, as passphrases still have not become popular among the general user population. Currently though there are several different ways of attacking pass-phrase creation strategies using context free grammars.

The first method would be to take a "Mad-Libs" approach, where additional replacements are used to designate specific input dictionaries are added to the base structures. For example a base structure could take the form:

Proper-Noun Verb a Noun -> Bob Cracks a Password (13)

Specific dictionaries can then assigned to common word types such as verbs, nouns, adjectives, etc. Each category may even have multiple dictionaries with different probabilities assigned to them. This means there might be a proper-noun dictionary targeted specifically to a certain user with the names of their friends and family assigned a very high probability and a larger dictionary of general names with a lower associated probability.

Note, that the original grammar replacements may also be used. Therefore the base structures still may contain S_n and D_n variables to represent the punctuation used. Also, pass-phrase specific

mangling strategies can also be modeled as variable replacements. For example, instead of using the whole sentence, a guess could be replaced by the first letter of each word, turning the above example, “Bob Cracks a Password”, into “BCaP”. Also certain words may be replaced by common substitutions, such as the word “one” being replaced by the number “1”. These replacements would all occur based on the probability they are encountered in the training set.

A slightly simpler approach would be to use an input dictionary composed entirely of commonly used pass-phrases. In this case the advanced word mangling rules described above may still be employed, but it would reduce the search space covered since the grammar is no longer generating arbitrary sentences. A final approach would be to build the entire sentence using a context free grammar, instead of basing it on a set base-structure “Mad Libs” approach. The main worry about that is depending on the grammar used, the search space could be extremely large. Therefore a lot of care would have to be put into how the grammar was structured.

CHAPTER 6

EVALUATING ENTROPY AS A METRIC FOR THE STRENGTH OF PASSWORDS

6.1 The NIST Model of Password Entropy

It is unlikely any other document has been as influential in shaping password creation and use policies as the NIST Electronic Authentication Guideline SP-800-63-1 [55]. The findings and recommendations published in it have proven the basis for many government and private industry password policies [56]. Central to this document is the notion of measuring password entropy. The idea of information entropy was first formalized by Claude Shannon [57], as a way of measuring the amount of information that is unknown due to random variables. In a way, it attempts to determine the randomness of a variable based upon knowledge contained in the rest of the message. Most often this "randomness" is expressed using the following equation:

$$\mathbf{H(x) = -\sum_{i=0}^{n} P(x_i)\, Log_2\, P(x_i)} \quad \textbf{(14)}$$

While other bases for the Log value may be chosen, when dealing with password entropy, base 2 is almost always selected due to the ability to represent the randomness prevalent in the system as equivalent to a random binary key of length H(x). For example a fair coin flip would land as heads 50% of the time. The resulting entropy of this would be $-\sum_{i=0}^{1} \mathbf{P}(.5)\, \mathbf{Log_2}\, \mathbf{P}(.5)$ which is equal to 1 bit of entropy. Each successive flip of the coin would add an additional bit of entropy as the result is a summation across all of the variables. In addition, if multiple variables are being modeled, the information leakage can be modeled as:

$$\mathbf{H(x,y) \leq H(x) + H(y)} \quad \textbf{(15)}$$

Due to the above properties, if the correct probability information is used, a defender can model the information leakage as the difficulty of an attacker guessing a random binary key of length H(x). The benefit this provides is the defender can then estimate the likelihood of an attacker, given a set number of guesses, can discover the correct value for the unknown variables. For example, the NIST SP-800-

63-1 document lists an acceptable number of attempts to allow an attacker to guess a random binary value for two different security levels:

Level 1: Number of Allowed Attempts = $2^{H(x)} \times 2^{-10}$ (1 in 1024 chance of compromise)
Level 2: Number of Allowed Attempts = $2^{H(x)} \times 2^{-14}$ (1 in 16,384 chance of compromise)

The above approach is labeled the *guessing entropy* of a system. The appeal of using the guessing entropy when designing password policies is obvious. If you can estimate the probabilities inherent in how users create passwords, you can then enforce policies that limit the chance of a successful password cracking attack to fall within acceptable levels. The difficulty in this approach is that estimating the information leakage caused by user selected passwords is not a trivial problem. If the entropy is significantly underestimated, (such that the resulting value for H(x) is considerably lower than the actual value), the resulting password policies may become overly burdensome to the users. Likewise if the entropy is overestimated, the security provided by the resulting policies may not be enough to protect against attacks.

In addition, as brought up in [58], the guessing entropy as detailed above does not provide a true measurement of the security provided by the system, as it does not take into account the number of targeted users, or the underlying distribution of their password selection. A proposed solution was detailed in [58], but only applied to binary passwords that are not directly generated by users, and as the author admits, is only applicable for uses such as one time passwords. Therefore the NIST model still remains the only widespread model of the security provided by user selected passwords.

Ignoring the weaknesses inherent in only using the guessing entropy, another question is how does the NIST model estimate the information leakage caused by users generating their own passwords under existing password creation rules? To that end the NIST paper details a set of rules that take advantage of the additive properties detailed in Equation 15 to derive a measurement of average entropy. These rules, quoted directly from the document, are as follows:

1. The entropy of the first character is taken to be 4 bits;
2. The entropy of the next 7 characters are 2 bits per character; this is roughly consistent with Shannon's estimate that "when statistical effects extending over not more than 8 letters are considered the entropy is roughly 2.3 bits per character;"
3. For the 9th through the 20th character the entropy is taken to be 1.5 bits per character;
4. For characters 21 and above the entropy is taken to be 1 bit per character;

5. A “bonus” of 6 bits of entropy is assigned for a composition rule that requires both upper case and non-alphabetic characters. This forces the use of these characters, but in many cases these (sp) characters will occur only at the beginning or the end of the password, and it reduces the total search space somewhat, so the benefit is probably modest and nearly independent of the length of the password;
6. A bonus of up to 6 bits of entropy is added for an extensive dictionary check. If the Attacker knows the dictionary, he can avoid testing those passwords, and will in any event, be able to guess much of the dictionary, which will, however, be the most likely selected passwords in the absence of a dictionary rule. The assumption is that most of the guessing entropy benefits for a dictionary test accrue to relatively short passwords, because any long password that can be remembered must necessarily be a “pass-phrase” composed of dictionary words, so the bonus declines to zero at 20 characters.

There has been some confusion based on the examples given in the NIST document whether rule #5 requires numbers and special characters to both be present or if the presence of either one would allow assigning of the “bonus” six bits of entropy. In Chapter 6.2, if such a distinction is important in any of the tests, the method to calculate rule #5 will be explicitly stated.

As an example of using the above model , consider a password creation policy requiring nine character passwords, and for at least one uppercase letter, lowercase letter, digit, and special character to be present. Fig 6.1.1 shows the calculation of the NIST entropy value.

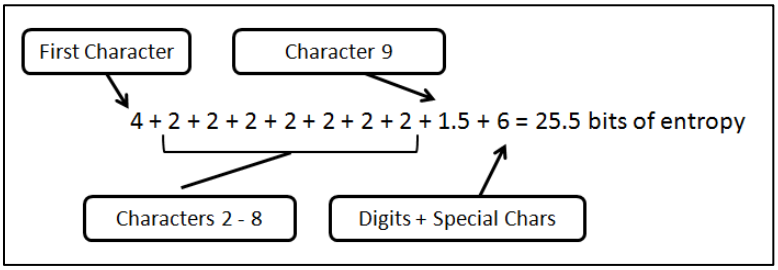


Figure 6.1.1: Calculation of the Guessing Entropy of a Password Creation Policy

Therefore, to limit an attacker to a guessing entropy value that provides enough security to meet the requirements of the level 1 classification, the number of allowed guesses would have to fall below:

$$2^{25.5} \times 2^{-10} = 2^{15.5} \approx 46{,}000 \text{ guesses} \quad (16)$$

6.2 Evaluating the NIST Model of Entropy against Password Cracking Attacks

The real question then is if the NIST model of entropy effectively predicts the resilience of a system against online password cracking attacks. The authors of the NIST publication themselves state, *"Unfortunately, we do not have much data on the passwords users choose under particular rules"*
To that end, I simulated several different password cracking attacks against the RockYou password test sets detailed in Chapter 1.6.5. First though, let's take a look at the distribution of passwords which could be created under different password policies based on the NIST entropy model. For this test, the minimum "entropy value" for each password was evaluated individually using the rules NIST put forward, with Rule 5 being interpreted as requiring both number and special characters. The reason "entropy value" is in quotation marks is that this value isn't the Shannon entropy value, but instead the minimum calculated NIST measurement at which that each password could be created under. The resulting values for entropy measurements ranging from 4 to 32, using RockYou test sets 1-3 can be seen in Fig. 6.2.1:

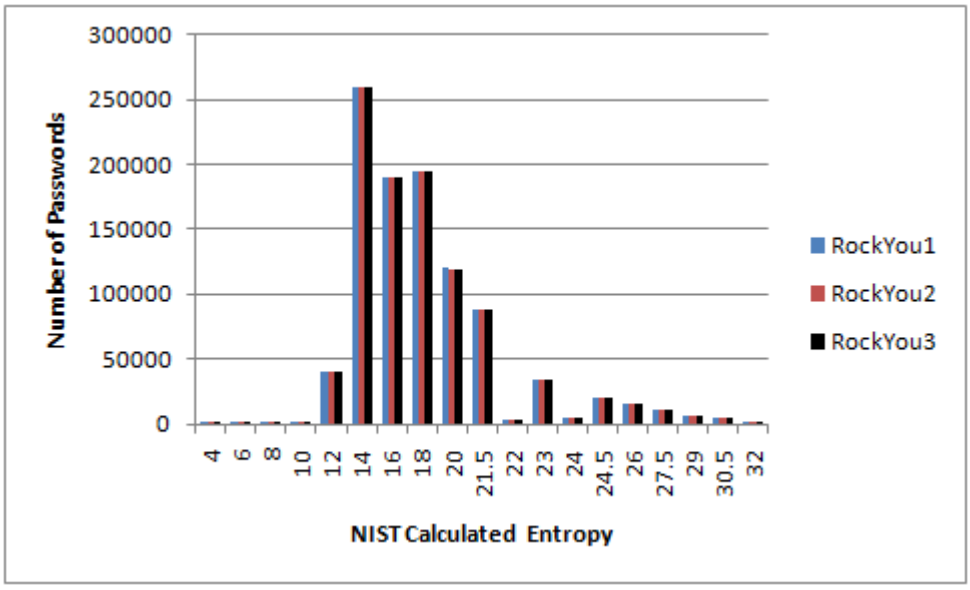


Figure 6.2.1 NIST Entropy Measurements for the RockYou Test Lists

The above graph may be a little difficult to read since the distribution of the NIST calculated entropy values was almost identical across the different test sets. In fact, the same test was run on the RockYou4, and RockYou5 test lists as well, but they were not included in the above graph since their results were the same. The lack of variation in itself is an interesting observation. This means that the

distribution above is likely to hold across other test sets as well when the test set, like the RockYou list, is generated without any external password creation policy being in place.

The next question is if increasing the minimum password length corresponds with adding 2 or 1.5 bits of uncertainty to the guessing entropy value of the resulting password creation policy. The reason why this is a concern is that most people create passwords based on dictionary words. Likewise in an online attack, the attacker is likely to employ a dictionary based guessing method as that increases their chances of cracking a password in a limited number of guesses. To simulate a generic dictionary based attack, John the Ripper [23], operating in wordlist mode using the single mode rule-set was employed. For the input dictionary, Dic-0294 [50] was selected. Multiple cracking sessions were then run against subsets of the RockYou1 test list, where each subset only contained passwords meeting a minimum length requirement. To further optimize the attack and to model the fact that an attacker may have knowledge of the password creation policy, each cracking session was only allowed to generate password guesses that meet the minimum length requirement as well. The results of cracking passwords with a min lengths ranging from seven characters to ten characters and limiting the attacker to one billion guesses can be seen in Fig. 6.2.2:

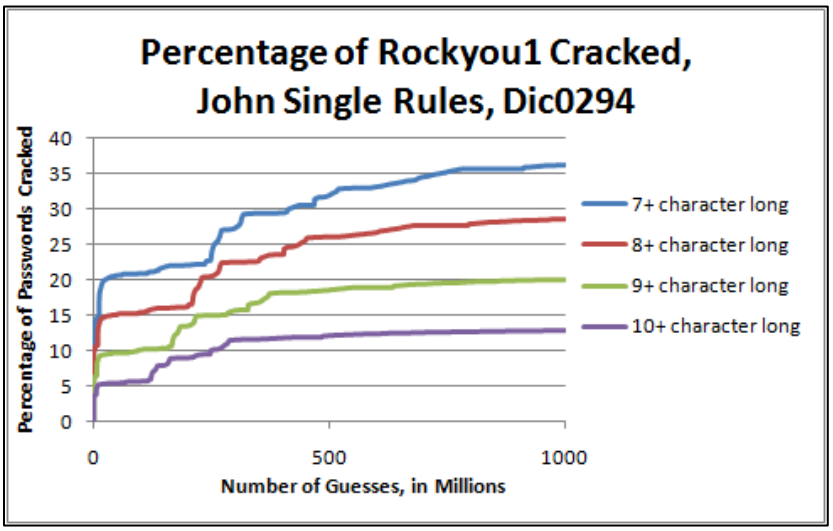


Figure 6.2.2: Cracking Attacks against Password Sets with Different Min Length Requirements

As can be seen, it appears the effectiveness of each attack was influenced by the minimum password length. Before we analyze how much security increasing the minimum password length provided in the above test, it is useful to discuss several other possible causes that might have caused this decrease in password cracking effectiveness. One thing to keep in mind is that the RockYou list did not enforce a password creation policy. This might lead to a subdivision of the security minded users.

For example, people who created longer passwords when they don't need to, (based on policy), might inherently choose a stronger password than users who create the simplest password allowed. Since none of the test sets I have access to enforced a password policy requiring users to create passwords longer than six characters long, verifying if this occurs is problematic. That being said, a statistical breakdown of the composition of the different sub-lists can be seen in Table 6.2.1:

Table 6.2.1: Composition of the Different RockYou1 Sub Lists

Character Set	7+ Chars	8+ Chars	9+ Chars	10+ Chars
Contains Digits	57.5%	59.5%	60.2%	60.0%
Contains Special Characters	4.4%	5.1%	6.6%	8.0%
Contains Uppercase	6.5%	6.7%	6.9%	7.1%
Contains Only Lowercase Letters and Digits	89.2%	88.4%	86.7%	85.1%

As expected, as shorter passwords were excluded, the average complexity of the remaining sets increased. What's interesting is that the percentage of uppercase letters rose slightly as the minimum password length was increased. Since uppercasing a letter does not change the length of a password, (vs. appending a digit), applying this mangling rule indicates that the users creating longer passwords were more concerned about security. Still, the percentage of users selecting uppercase characters remained low, and the above statistics do not seem to fully account for the difference in resistance to password cracking attacks across the different sub-lists. To further confirm this, the above lists were once again sub-divided to only include passwords containing at least one digit. Likewise, the password cracking attacks were further modified so they only produced guesses that contained digits as well. The results of re-running the password cracking attacks against these new lists can be seen in Fig. 6.2.3.

The test shows, even when all passwords were required to contain digits, there still was a noticeable difference in the effectiveness of the attacks against the different test sets. That being said, the divergence between the test sets was less than if no digits were required; The attack against the 7+ character set performed worse when digits were required, and the attack against the 10+ character set surprisingly performed slightly better when digits were required. Note: it would probably be the wrong conclusion to assume that requiring digits in a password creation policy would make 10 character long passwords weaker. What more likely happened is that this additional rule excluded "junk" passwords in

the list that did not correspond to an actual user generated password, (for example some URLs greater than a 100 characters were present in the RockYou list). In addition several passwords may have been created in a way that made them strong but did not include digits, such as certain pass-phrases. By excluding digits, some of these strong passwords may have been excluded as well.

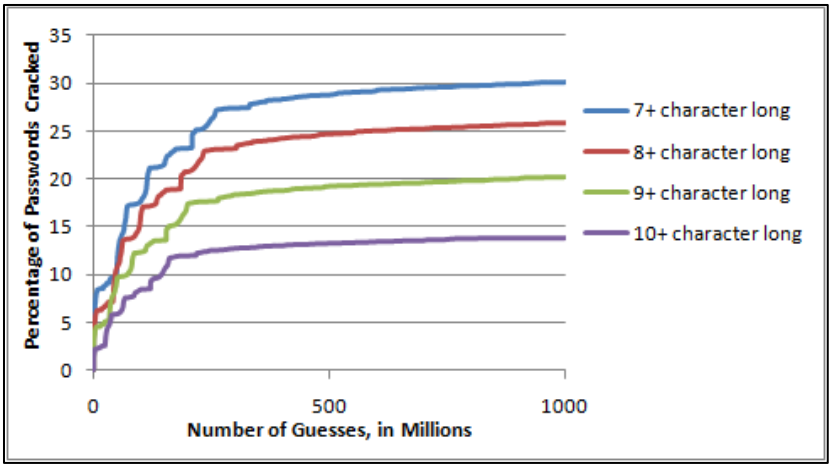


Figure 6.2.3: Cracking Passwords Sets of Different Lengths Which Contained Non-Lowercase Letters

The next step is to evaluate what effect increasing the minimum password length has on the effectiveness of a password cracking attack. To accomplish this, we must first determine a relevant method to compare the success of different password cracking attacks to each other. This is not a trivial problem. One approach would be to compare the final number of passwords cracked. A problem with that approach is: how long should the cracking session be limited to? As an example of that, Fig 6.2.4 shows the same cracking sessions detailed in Fig. 6.2.3, but this time limited to 50 thousand guesses.

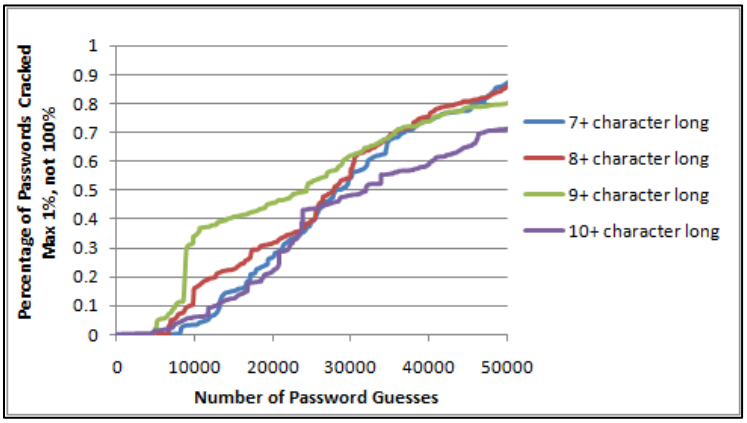


Figure 6.2.4: A Shorter Password Cracking Session of Passwords that Contained Digits

This time, the results are decidedly mixed with the minimum length only making a noticeable difference in the effort required to crack 10+ character long passwords. It also is worth keeping in mind that the above attack was not optimized for a password cracking session of that length, as the input dictionary itself, dic-0294, contained over 800 thousand words. It is interesting though that even in this un-optimized attack, each of the cracking sessions nearly cracked 1% of the total passwords. We'll see in the next test though that 1% cracked over 50,000 guesses is actually a poor success rate. Another method would be to determine how many guesses were required to crack a certain percentage of passwords. Once again though, this is highly dependent on the limit set, as be seen in the previous graph 6.2.3, as a password cracking session starts out fast, but quickly slow down as it takes more and more guesses to crack each successive password.

As briefly mentioned, the above attack wasn't optimized for a limit of 50,000 guesses. Considering the NIST 800-63-1 document is primarily concerned with online password cracking attacks, it is very important to evaluate the effectiveness of shorter password cracking sessions. To set the next test up, I combined the training password lists RockYou28 through RockYou32 to create a training list of five million user passwords. I then sorted the entire training list by the number of occurrences of each password, (so the first password was '123456'), and then removed all duplicates. This resulted in an input dictionary where the most probable guesses are tried first. I then used it as input to generate 50,000 guesses against the various sub-lists, generated from the RockYou1 test list, that were used in the previous examples. The results of this attack when limited to 50,000 guesses can be seen in Fig 6.2.5:

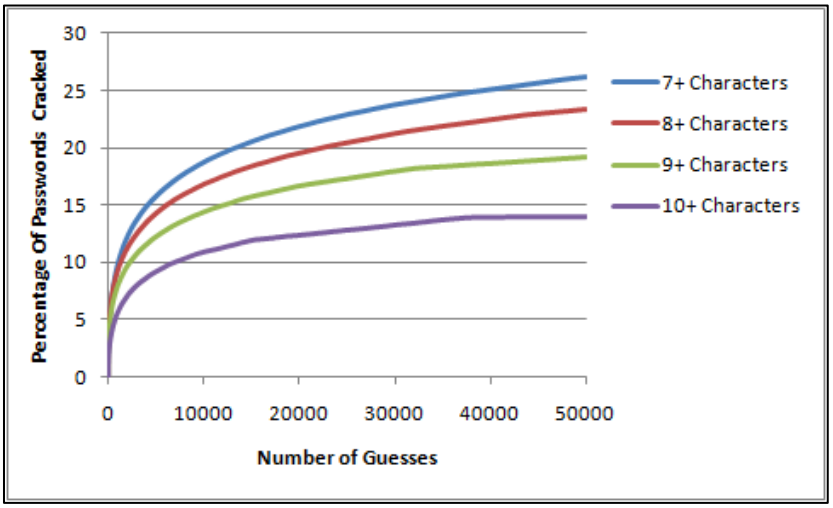


Figure 6.2.5: Cracking Sessions against the RockYou1 Test Lists, Targeted Against 50,000 Guesses

The interesting thing about the above graph is given just fifty thousand guesses, over 25% of the 7+ character long passwords were cracked, and over 14% of the 10+ character passwords were cracked. This begs the question, how does the above cracking session compare to the acceptable NIST guessing values given for level 1 and 2 systems? The answer, based on the above tests can be seen in Table 6.2.2:

Table 6.2.2 Comparison of a Targeted Cracking Attack against the NIST Guessing Values

Value	7+ Chars	8+ Chars	9+ Chars	10+ Chars
NIST Entropy	16	18	19.5	21
Level 1 # of Guesses	64	256	724	2048
% Cracked Using Guesses Allowed by Level 1	3.21%	6.04%	7.19%	7.12%
Acceptable Level1 Failure Rate	0.097%	0.097%	0.097%	0.097%
Level 2 # of Guesses	4	16	45	128
% Cracked Using Guesses Allowed by Level 2	0.98%	2.19%	2.92%	2.63%
Acceptable Level2 Failure Rate	0.0061%	0.0061%	0.0061%	0.0061%

As can be seen, if a blacklist of common passwords is not part of the password creation policy, the NIST model quickly breaks down. For comparison, a policy that meets a Level 1 system should only allow an attacker to guess a password within the allowed number of guesses with a probability of 1 in 1024, or approximately a 0.097% chance. Likewise, a Level 2 certified system should only allow an attacker to be successful with a probability of 1 in 16,384, or approximately a 0.0061% chance. This is drastically different from the above findings. What's worse, the attacker's success rate actually increases as the number of guesses allowed grows due to the higher minimum password length, (though it does drop a bit when 10+ characters are required). This implies that the current NIST measurements overestimate the security provided by increasing the minimum password length.

In all fairness, in the NIST 800-63-1 document, it does state that a blacklisting approach was required for a minimum entropy estimate. That slightly contradicts though their original entropy calculation, where they assign an optional 6 bits of guessing entropy if a blacklist is employed. What the

above test shows is that a blacklist approach must be employed for any sort of security against online password cracking attacks, considering 1.74% of 7+ character long passwords were cracked with just sixteen guesses.

The next question is how effective are different sized blacklists of prohibited passwords. This was simulated by creating blacklists based on the training list input dictionary from the previous test. For example, a blacklist containing 500 banned passwords would be formed from the 500 most frequently used passwords in the training set. The results of re-running the attacks depicted in Fig. 6.2.5 against the RockYou1 test list containing 7+ character words, and using various length blacklists, can be seen in Fig 6.2.6. It should also be noted that for these password cracking sessions, it was assumed that the attacker would have knowledge of the blacklist of banned passwords, so they would not use them in their attacks, (aka for the 500 banned password blacklist, the attacker would start by guessing the 501'th most probable password guess).

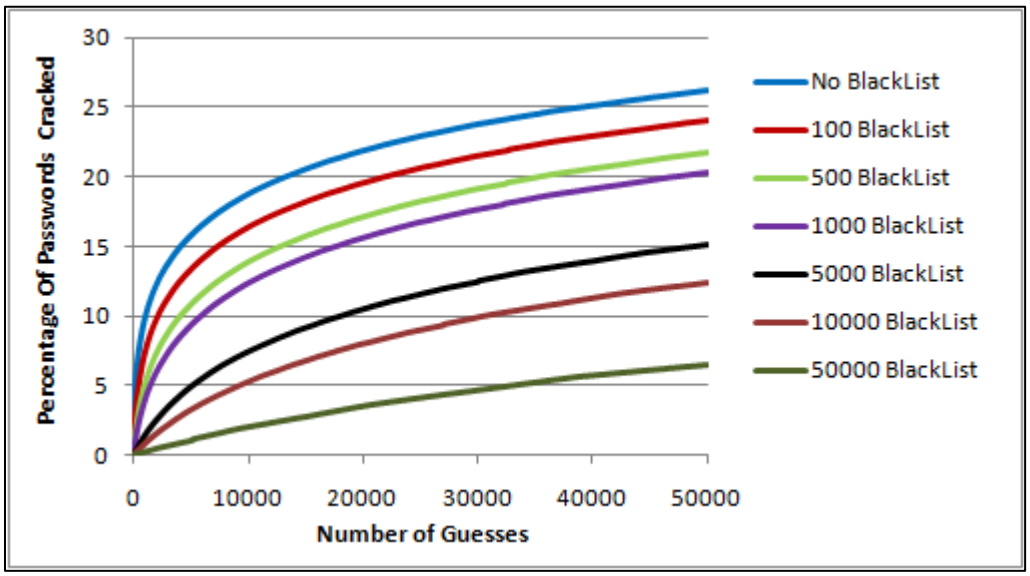


Figure 6.2.6 BlackLists Used to Defend Against Online Attacks against the RockYou1 Test List

The most restrictive password creation policy in this test, where passwords must be seven characters long and a blacklist of 50,000 words is used, yields a NIST calculated entropy value of 22. For a Level 1 certified system, that means an attacker would be able to make 4096 guesses, and for a Level 2 system the attacker would be able to make 256 guesses. Digging into the results depicted in Fig 3.6, this would lead to an attacker cracking 0.848% of the passwords for a Level 1 certified system, and 0.058% of the passwords for a Level 2 certified system. While this is a much higher failure rate

than estimated by the NIST model, it represents a definite improvement over the results when a blacklist is not employed.

Now a legitimate concern with the previous tests is that while the training passwords and the test passwords were different, they both came from same website. It can be argued that this represents an unfair advantage for the attacker, and would not translate to a real world password cracking session. To that end, using the same dictionary formed from the RockYou training set, password cracking sessions were run against several other disclosed password lists using a maximum of 50k guesses.. As in the previous tests, these lists were divided by minimum password length. The results of these password cracking attacks can be seen in Fig. 6.2.7.

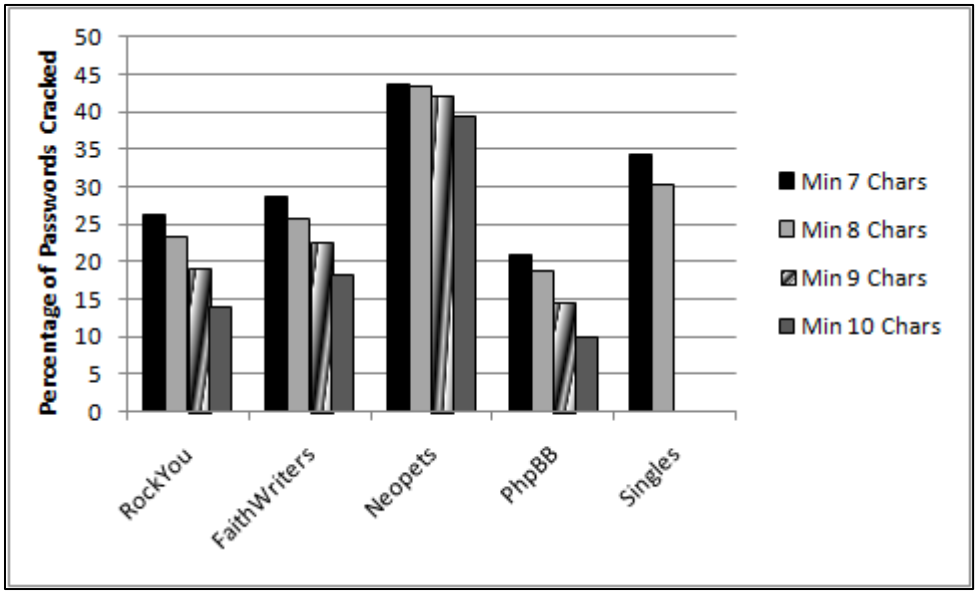


Figure 6.2.7 Online Cracking Session using 50k Guesses against Various Password Lists

As Fig. 6.2.7 shows, the cracking sessions against the RockYou dataset actually performed worse than most of the other datasets. Much of this is probably due to user training, and the relative importance of the passwords to the users. For example, the Neopets list for the most part represents young children, while the PhpBB list was a development and distributions site for the PhpBB bulletin board which means it can be assumed that most of its users were webmasters and/or programmers.

Another point of interest is the effectiveness of the RockYou training set dictionary compared with other available input dictionaries. To illustrate this, several cracking sessions were run against the FaithWriters password list. Since the FaithWriters and the Singles.org password lists were both composed of people almost exclusively of the Christian faith, another targeted input dictionary was created from the Singles.org list. This dictionary was generated in the same way as the dictionary from

the RockYou training list. Since Singles.org represented a much smaller password list, it only contained 12.234 unique words. Likewise the default dictionary provided with John the Ripper, passwords.lst was also used. Due to password.lst's very small size, (3,116 words), the default John the Ripper mangling rules were also applied to generate some of the extra guesses required. All three input dictionaries were then allowed to make 10 thousand password guesses against the FaithWriters list. The results of this can be seen in Fig. 6.2.8.

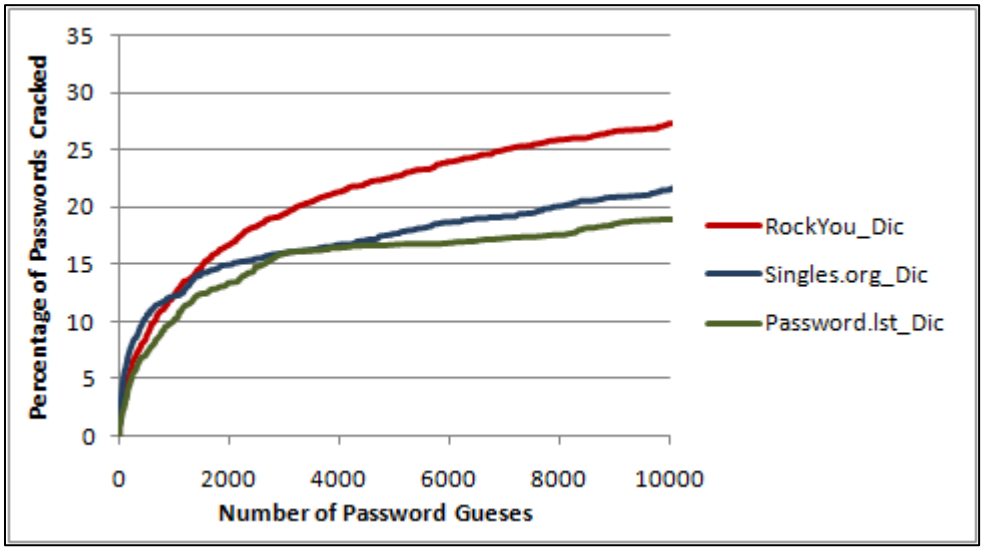


Figure 6.2.8: Multiple Targeted Dictionaries against the Singles.org List

As expected, the Singles.org dictionary performed slightly better initially, but it was too small of a sample size to compete with the custom RockYou dictionary over a longer password cracking session. Both custom dictionaries though performed better than the default John the Ripper dictionary. This test illustrates though that the findings depicted in this chapter can always be improved on as an attacker gains more knowledge about their targets.

The main lesson learned from these experiments is that the NIST 800-63 defined entropy does not correlate to the actual guessing entropy of a test set of passwords. While this has been pointed out in previous works [79, 80], the previous tests show it empirically as well. Furthermore, these experiments imply that a method for rejecting weak passwords, such as a blacklist approach, may provide much more security against online attacks than specifying base rules, (such as password length), during password creation.

CHAPTER 7

CONCLUSION

When originally starting this research, I was skeptical about my chances of finding new and novel techniques to enhance password cracking attacks. Password cracking has been around ever since someone invented the first secret word. What I didn't expect was until recently, due to the sensitive nature of passwords, very few datasets were publicly available to develop and evaluate innovative cracking techniques on. Therefore I was fortunate to conduct my research at a time when the widespread adoption of stronger password storage techniques combined with publicly disclosed password lists have spurred new explorations into this field.

I hope this paper brings forward the idea that by applying advanced modeling techniques, the effectiveness of a password cracking attack can be drastically increased. While individual users can always create passwords that are resistant to these models, the effectiveness of these attacks is based on the fact that users as a whole have a hard time being unique. This means that even in the future, if users adopt new ways to create passwords, an attacker will most likely still be able to model the new creation methods as well.

Going forward, I fully expect the probability methods described in this paper to become more prevalent in password cracking attacks. I'm especially proud of the work detailed in Chapter 5 on using probabilistic context free grammars to model user password creation. This is a completely new technique, and one that will only increase in effectiveness as additional optimizations and improvements are added to it. In addition, by allowing attackers to pre-compute dictionary based attacks as described in Chapter 4 and re-use the results later, highly advanced word mangling rules and large dictionaries can be incorporated into normal password cracking sessions. This is in contrast to before, where often due to time constraints, this was not possible.

That being said, I remain cautiously optimistic about the future of human memorable passwords. The key though is to enhance the underlying security in how passwords are stored, accessed and the policies that govern them, rather than continuing to blame users for poor password selection. Techniques such as blacklisting common passwords and limiting the number of password guesses allowed to an attacker can drastically increase the cost and decrease the effectiveness of password cracking attacks. In addition, widespread adoption of practices such as using password salts can effectively eliminate some classes of attacks such as hash pre-computation. While perfect password security will never be a reality, the future is not as dire as it might seem.

APPENDIX A

PSUEDO-CODE FOR THE PROBILISTIC PASSWORD CRACKER NEXT FUNCTION

```
//The probability calculation depends on if pre-terminal or terminal probability is used
//New nodes are 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)
}
```

APPENDIX B

PSUEDO-CODE FOR THE PROBILISTIC PASSWORD CRACKER DEADBEAT DAD FUNCTION

```
//The probability calculation depends on if pre-terminal or terminal probability is used
//New nodes are 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.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=0; i<working_value.num_strings;i++) {
             child_node=decrement(working_value.structure,i);  //get next lower probability S or D structure at
             //pivot value 'i'
             child_node.num_strings = working_value.num_strings
            if ((child_node!=NULL) AND (lowest_probability_parent(child_node, working_value,i)) {
                        child_node.probability = calculate_probability(child_node.structure);
                        insert_into_priority_queue(child_node,insert_value)
            }
        }
        working_value = Pop(priority_queue)
}


Boolean function lowest_probability_parent(node child_node, node working_value, int parent_pivot) {
        For (i=0;i<child_node.num_stings;i++) {
                If (i!=parent_pivot) {  //not the parent node
                        //find the probability of the other parent
                        other_parent_probability = generate_parent_probability(child_node, i);
                        if (other_parent_probability<working_value.probability) {
                                //some other parent will take care of the child
                                Return false;
                        }
                        else if (other_parent_probabity == working_value.probability) {
                                //the location of the pivot is used to break ties
                                if (i>parent_pivot) {
                                        Return false;
                                }
                        }
                }
        }
        Return true;
}
```

LIST OF REFERENCES

[1] D. Goodin. “Crypto spares man who secretly video taped flatmates,” The Register, October 21st, 2009, [Online Document] [cited 2010 Feb 17] http://www.theregister.co.uk/2009/10/21/flatmate_spy_encryption_case/

[2] R. McMillan, “Phishing attack targets Myspace users”, 2006, [Online Document] [cited 2010 January 14] Available HTTP http://www.infoworld.com/d/security-central/phishing-attack-targets-myspace-users-614

[3] Batmud Administrators, “Batmud Hacked”, October 2007, [Online Document] [cited 2-10-2010] Available HTTP http://www.bat.org/node/32

[4] R. Ferguson, “ZF05, Kaminsky = 0wned, Mitnick = 0wned”, July 29, 2009, [Online Document] [cited 2-10-2010] Available HTTP http://countermeasures.trendmicro.eu/zf05-kaminsky-0wned-mitnick-0wned/

[5] T. Warren, “Thousands of Hotmail Passwords Leaked” [Online Document] [cited 2010 January 14] Available HTTP http://www.neowin.net/news/main/09/10/05/thousands-of-hotmail-passwords-leaked-online

[6] The Pirate Bay [Online Site] [cited 2010 January 14] Available HTTP http://www.thepiratebay.org

[7] A. Vance, “If Your Password is 123456 Just Make it HackMe” New York Times, January 20th, 2010. Page A1

[8] BayWords Mission Statement. [Online Document] [cited 2-19-2010] Available HTTP http://baywords.com/about

[9] DarkC0de Website. [Online Document] [cited 2-19-2010] Available HTTP http://www.darkc0de.com

[10] HashKiller Website. [Online Document] [cited 2-19-2010] Available HTTP http://www.hashkiller.com

[11] Inside Pro Password Recovery Site [Online Document] [cited 2-19-2010] Available http://www.insidepro.com/

[12] MD5-Utils Online Password Cracker [Tool] [cited 2-19-2010] Available http://md5-utils.sourceforge.net/

[13] Wikipedia, “Crypt (Unix)” [Online Document] Available http://en.wikipedia.org/wiki/Crypt_(Unix)

[14] Baldwin, B, “Crypt Breaker’s Workbench”, [Software] written 1984. Available http://axion.physics.ubc.ca/cbw.html

[15] N, Provos, and D. Mazières, "A Future-Adaptable Password Scheme". Proceedings of 1999 USENIX Annual Technical Conference: 81–92,

[16] Wikipedia, “LM Hash” [Online Document] Available http://en.wikipedia.org/wiki/LM_hash

[17] M. Burnett, “Ten Windows Password Myths” [Online Document] Available http://www.securityfocus.com/infocus/1554

[18] M. Weir and S. Aggarwal. “Cracking 400,000 Passwords or How to Explain to Your Roommate why the Power-Bill is a Little High”, Defcon 17, Las Vegas, NV, August 2009

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

[20] Renderman, “WPA-Rainbow Tables” [Online Document and Code] Available http://www.renderlab.net/projects/WPA-tables/

[21] Access Data [Software] PRTK, [cited 2-19-2010] Available HTTP http://www.accessdata.com/decryptionTool.html#passwordrecoverytoolkit

[22] ElcomSoft, [Software] ElcomSoft Distributed Password Recovery [cited 2-19-2010] Available HTTP http://www.elcomsoft.com/products.html

[23] The OpenWall Group, [Software] John the Ripper password cracker, [Online Document] [cited 2-19-2010] Available HTTP http://www.openwall.com

[24] Mao, [Software] Cain and Able [Online Document] [cited 2-19-2010] Available HTTP http://www.oxid.it

[25] Shuanglei, Zhu, “Project RainbowCrack”, [Online Document] http://project-rainbowcrack.com/, retrieved March 07, 2009

[26] rcracki, [Online Document and Tool] http://www.freerainbowtables.com/, retrieved March 07. 2009

[27] Zim, Herbert Spencer. Codes and secret writing (abridged edition). Scholastic Book Services, fourth printing, 1962. Copyright 1948 Herbert S. Zim. Originally published by William Morrow.

[28] Kuo, C. Romanosky, S. Cranor, L. “Human Selection of Mnemonic Phrase-based Passwords” Symposium on User Privacy and Security, July 2006, Pittsburg, PA

[29] J. Yan, “A Note on Proactive Password Cracking”, Proceedings of the 2001 workshop on New security paradigms, Pages 127-135, 2001, Cloudcroft, New Mexico

[30] L. Stitson, Intro to Cryptography Notes, Stanford University [Online Document] [cited 2-19-2010] Available HTTP http://www.stanford.edu/~stinson/crypto/S3995/class_3.txt

[31] Z. Grunschlag, “Cryptanalysis with WebCrypt”, Columbia University Slides [Online Document] [cited 2-19-2010] Available HTTP http://www1.cs.columbia.edu/~zeph/software/webcrypt/webcryptanalysis.pdf

[32] 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

[33] M. Weir, [Software] MiddleChild Password Cracker, [cited 2-19-2010] Available HTTP http://sites.google.com/site/reusablesec/Home/password-cracking-tools/middle-child

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

[35] 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.

[36] Paul C. van Oorschot, Michael J. Wiener: Parallel Collision Search with Application to Hash Functions and Discrete Logarithms. ACM Conference on Computer and Communications Security 1994: pp210–218

[37] Weir, Matt, [Sofware] “Drcrack Dictionary Based Rainbow Tables”, [cited 2-19-2010] Available HTTP, http://sites.google.com/site/reusablesec/Home/rainbow-tables.

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

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

[40] 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

[41] The Hacker’s Choice [Software] “THC Hydra”, [cited 2-19-2010] Available HTTP, http://www.thc.org/thc-hydra/

[42] The NCrack Development Group, [Software] “NCrack password cracker”, [Online Document] [cited 2-19-2010] Available HTTP http://nmap.org/ncrack/

[43] A. Forget, S. Chiasson, P.C. van Oorschot, R. Biddle, “Improving Text Passwords Through Persuasion.” *Symposium on Usable Privacy and Security (SOUPS) 2008*, July 23–25, 2008, Pittsburgh, PA USA

[44] L0pht Heavy Industries, [Software], “L0phtCrack Password Cracker”, [Online Document] [cited 2-19-2010] Available HTTP http://www.l0phtcrack.com/

[45] R. Raraine, D. Danchev “PhpBB Hacked”, [Online Document] [cited 2010 January 14] Available HTTP http://blogs.zdnet.com/security/?p=2493

[46] M. Weir, S. Aggarwal, B. De Medeiros, B. Glodek. “Password Cracking Using Probabilistic Context-Free Grammars”. In Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (May 17 - 20, 2009). SP. San Fransico, CA, 391-405.

[47] User bofh28, [Software], “Crunch Password Generation Tool”, [Online Document] [cited 2-19-2010] Available HTTP http://crunch-wordlist.svn.sourceforge.net/

[48] B. Glodek, “Using a Specialized Grammar to Generate Probable Passwords”, Thesis, Florida State University, September 2008, Tallahassee FL.

[49] M. Weir. [Software], “Probabilistic Password Cracker” [cited 2010 January 14] Available HTTP http://sites.google.com/site/reusablesec/Home/password-cracking-tools/probablistic_cracker

[50] *A list of popular password cracking wordlists*, 2005, [Online Document] [cited 2010 January 14] Available HTTP http://www.outpost9.com/files/WordLists.html

[51] Solar Designer, “John the Ripper Modes README” [Online Document] [cited 2010 January 14] Available HTTP http://www.openwall.com/john/doc/MODES.shtml

[52] The Wikipedia Foundation, “Wiktionary.org”, [Online Document] [cited 2010 January 14] Available HTTP http://www.wiktionary.org

[53] S. Raveau, “Cracking Passwords with Wikipedia, Wiktionary, Wikibooks, etc”, [Online Document] [cited 2010 January 14] Available HTTP http://blog.sebastien.raveau.name/2009/03/cracking-passwords-with-wikipedia.html

[54] Navarro G (2001). "A guided tour to approximate string matching". ACM Computing Surveys 33 (1): 31–88

[55] W. Burr, D. Dodson, R. Perlner, W. Polk, S. Gupta, E. Nabbus, “NIST Special Publication 800-63-1 Electronic Authentication Guideline”, Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, December 8, 2008

[56] Office of Management and Budget, “Draft Agency Implementation, Guidance for Homeland Security, Presidential Directive 12”, August 2004.

[57] C.E. Shannon, "A Mathematical Theory of Communication", Bell System Technical Journal, vol. 27, pp. 379-423, 623-656, July, October, 1948

[58] E. R. Verheul. “Selecting secure passwords”, CT-RSA 2007, Proceedings Volume 4377 of Lecture Notes in Computer Science, pages 49–66. Springer Verlag, Berlin, 2007.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

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

[64] 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.

[65] Wayman, J. “Fundamentals of Biometric Authentication Technology” International Journal of Image and Graphics (IJIG), Year 2001, Volume 1, Issue 1. Pages 93 – 113.

[66] Franken, Z. “Biometric and token based access control systems: Are you protected by two screws and a plastic cover? Probably.” DefCon 15 Presentation. August 2007, Las Vegas Nevada

[67] Merkle, R. “Protocols for public key cryptosystems” sp,pp.122, 1980 IEEE Symposium on Security and Privacy, 1980

[68] Weaver, A. "Biometric Authentication," Computer, vol. 39, no. 2, pp. 96-97, Feb., 2006

[69] Kwon, T. “Ultimate Solution to Authentication via Memorable Password” Contribution to the IEEE P1363 study group for Future PKC Standards

[70] Adams, A. Sasse M., “Users are Not the Enemy”, Communications of the ACM, Vol 42, Issue 12, Pages: 40-46, December 1999

[71] Thorpe, J. Oorschot, P. “Graphical Dictionaries and the Memorable Space of Graphical Passwords” 13th USENIX Security Symposium, August 9–13, 2004 San Diego, CA

[72] Zhuang, L. Zhou, F. Tygar, J. “Keyboard Acoustic Emanations Revisited” 12th ACM Conference on Computer and Communications Security, Pages 373-382, Alexandria, VA 2005

[73] Soghoian, Chris, Trukish police may have beaten encryption key out of TJ Maxx suspect, October 2008, [cited 2010 January 14] Available http://news.cnet.com/8301-13739_3-10069776-46.html

[74] Stephey, M. “Sarah Palin’s E-mail Hacked”, Time.com, Sep 17, 2008. [Online Document] [cited 2010 January 14] http://www.time.com/time/politics/article/0,8599,1842097,00.html

[75] Hansson, W. “Judge Says Man Can’t be Forced to Divulge PGP Encryption Password”, December 2007. [Online Document] [cited 2010 January 14] Available http://www.dailytech.com/article.aspx?newsid=10058

[76] Beck, M. Tews, E. “Practical attacks against WEP and WPA” November 8th, 2008. [Online Document] [cited 2010 January 14] http://dl.aircrack-ng.org/breakingwepandwpa.pdf

[77] D. Smith. “Forensic Image Analysis to Recover Passwords”, ShmooCon 2008, February 2008. Washington D.C.

[78] D. Feldmeier and P. Karn, “UNIX Password Security – Ten Years Later” Advances in Cryptology, CRYPTO ’89, 9th Annual International Cryptology Conferences, Santa Barbara, California, USA, August 20-24, 1989, Proceedings. Pages:44-63

[79] E. R. Verheul. “Selecting secure passwords”, CT-RSA 2007, Proceedings Volume 4377 of Lecture Notes in Computer Science, pages 49–66. Springer Verlag, Berlin, 2007

[80] J.L. Massey, “Guessing and Entropy,” Proc. 1994 IEEE International Symposium on Information Theory, 1995, p.329

[81] M. Weir, “Smarter Password Cracking”, ShmooCon 2008, February 2008, Washington D.C.

BIOGRAPHICAL SKETCH

Charles Matt Weir

Charles Matt Weir, known as Matt to his friends, has always been interested in security systems, both physical and digital, with his favorite book being “The Stainless Steel Rat”, by Harry Harrison. Graduating with a B.S. in Computer Science from Virginia Tech, it was actually his minors in Math and Philosophy, along with his involvement in the impromptu acting group the Camarilla, that have helped him the most. In 2004, Matt obtained his Masters in Information Security from Florida State University, and left the academic world for a job as a Network Security Engineer. The projects he worked on ranged from performing penetration tests to building wireless networks for first responders. In 2007 Matt set aside his job and traveled back to Florida State University to pursue a PhD degree in Computer Science. Beyond his research into password cracking, he is also interested in network security, computer forensics, and learning to skateboard.