FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

PROBABILISTIC CONTEXT-FREE GRAMMAR BASED PASSWORD CRACKING: ATTACK, DEFENSE AND APPLICATIONS

By

SHIVA HOUSHMAND YAZDI

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

2015

Shiva Houshmand Yazdi defended this dissertation on July 8, 2015.
The members of the supervisory committee were:

Sudhir Aggarwal
Professor Directing Dissertation

Washington Mio
University Representative

Piyush Kumar
Committee Member

Xin Yuan
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the dissertation has been approved in accordance with university requirements.

To the memory of my beloved grandmother
for her unconditional love and support

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to my advisor Professor Sudhir Aggarwal for his mentorship and guidance throughout all these years. I cannot thank him enough for his wisdom, patience and understanding. His enthusiasm for science and attention to mathematical details has taught me to think critically and independently. I would like to thank him for his constant support, advice, and encouragement throughout my graduate studies. For everything you have done for me, I thank you.

I am sincerely grateful to my doctoral committee members Dr. Xin Yuan, Dr. Piyush Kumar, and Dr. Washington Mio for their encouragement and valuable comments. Special thanks to Dr. Kumar, his guidance and assistance has helped me grow professionally.

I would also like to thank all my friends who provided support, inspiration and motivation along the way. I am thankful to my colleagues at Florida State University that made this experience joyful. In particular I am grateful to Umit and Randy that helped and supported me in this research work. Special thanks to my lifelong friends Azadeh, Noushin, Sanaz and Saba who have always been there for me regardless of the physical distance between us.

I would like to express my appreciation to my loving family for their love and support. Special thanks to my mother, for all the sacrifices she has made. She has empowered me with strength and courage to tackle any challenge. I am truly indebted to her for making it possible for me to pursue my dreams. To my eldest sister Shadi, for her constant support. Her determination in life has encouraged me to endure the most difficult times. To my sister Shirin, for being the most amazing friend and role model. Her hard work and determination in life has always motivated me to strive to be the best. Her love and confidence in me has helped me become the person I am today.

Most importantly I would like to thank my husband Omid, for his unconditional love, endless support, and infinite patience. He has made many sacrifices to be by my side every step of this journey, and has constantly comforted me, encouraged me and lifted me up whenever I was down. I love him dearly and greatly appreciate his wholehearted devotion and belief in me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Passwords are critical for security in many different domains such as social networks, emails, encryption of sensitive data and online banking. Human memorable passwords are thus a key element in the security of such systems. It is important for system administrators to have access to the most powerful and efficient attacks to assess the security of their systems more accurately. The probabilistic context-free grammar technique has been shown to be very effective in password cracking. In this approach, the system is trained on a set of revealed passwords and a probabilistic context-free grammar is constructed. The grammar is then used to generate guesses in highest probability order, which is the optimal off-line attack. The initial approach, although performing much better than other rule-based password crackers, only considered the simple structures of the passwords. This dissertation explores how classes of new patterns (such as keyboard and multi-word) can be learned in the training phase and can be used to substantially improve the effectiveness of the probabilistic password cracking system. Smoothing functions are used to generate new patterns that were not found in the training set, and new measures are developed to compare and improve both training and attack dictionaries. The results on cracking multiple datasets show that we can achieve up to 55% improvement over the previous system. A new technique is also introduced which creates a grammar that can incorporate any available information about a specific target by giving higher probability values to components that carry this information. This grammar can then help in guessing the user's new password in a timelier manner. Examples of such information can be any old passwords, names of family members or important dates. A new algorithm is described that given two old passwords determines the transformations between them and uses the information in predicting user's new password.

A password checker is also introduced that analyzes the strength of user chosen passwords by estimating the probability of the passwords being cracked, and helps users in selecting stronger passwords. The system modifies the weak password slightly and suggests a new stronger password to the user. By dynamically updating the grammar we make sure that the guessing entropy increases and the suggested passwords thus remain resistant to various attacks. New results are presented that show how accurate the system is in determining weak and strong passwords.

Another application of the probabilistic context-free grammar technique is also introduced that identifies stored passwords on disks and media. The disk is examined for potential password strings and a set of filtering algorithms are developed that winnow down the space of tokens to a more manageable set. The probabilistic context-free grammar is then used to assign probabilities to the remaining tokens to distinguish strings that are more likely to be passwords. In one of the tests, a set of 2,000 potential passwords winnowed down from 49 million tokens is returned which identifies 60% of the actual passwords.

CHAPTER 1

INTRODUCTION

Despite much research in newer authentication techniques such as biometric based techniques or graphic based authentication, passwords still remain the primary method for authentication. Passwords are critical for security in many different domains such as social networks, emails, encryption of sensitive data and online banking. Because of the fairly universal use of passwords, it is often necessary for law enforcement to be able to crack passwords and thus it has been important to make progress in cracking techniques. It is also important for system administrators to have access to the most powerful and efficient attacks to assess the security of their systems more accurately.

In an offline password cracking session, the attacker has already obtained the password hashes or encrypted files. Since the hash functions used to store passwords are one-way functions and cannot be easily inverted, the attacker repeatedly makes a password guess, applies the same hash algorithm to the guess and then compares it with the obtained hash to check whether they match or not. An important advance in password cracking was the work proposed by Weir et al. [1]. In this approach a probabilistic context-free grammar was used to generate guesses in highest probability order. This approach, although shown to be very effective compared to other password crackers, only considered the simple structures of the passwords and represented passwords simply as sequences of symbols, digits and alphabet characters.

The novel contributions in this dissertation can be categorized into the followings: (1) I improve the probabilistic password cracking technique of Weir et al. [1] by learning new classes of patterns (such as keyboard and multi-word patterns) in the training. I also develop new metrics for comparing and analyzing attack dictionaries and show that these techniques can improve the efficiency of password cracking by 55%; (2) I develop a new technique to perform targeted password cracking by incorporating the available information about the target into the probabilistic context-free grammar; (3) I then describe a system that can leverage from the knowledge gained from password cracking techniques and use it to estimate the strength of user chosen passwords and help users to create stronger passwords; and (4) I show another

application of the probabilistic context-free grammars by developing a system that aims to find stored passwords on disk and media in order to help investigators in digital forensics area.

In this dissertation I explore how a class of new patterns (such as keyboard and multi-word patterns) can be learned in the training phase and can be used systematically to continue cracking in highest probability order and to substantially improve the effectiveness of the password cracker. It was assumed that keyboard patterns result in strong passwords because they create seemingly random strings but can nevertheless be easily remembered. Multi-words (or passphrases) have been also widely proposed as a way to build stronger and more memorable passwords. They are also often used when longer passwords are required because they are supposedly more resistant to brute force attacks. These two main classes of patterns are used commonly in passwords and the typical approach to attack such patterns is to add a list of common patterns into an attack dictionary. In this dissertation I first identify the patterns in the training password list, and then learn the patterns by incorporating them into the probabilistic context-free grammar. I also explore the use of smoothing functions to generate new patterns that were not found in the training set.

In dictionary-based attacks, a list of words called an attack dictionary is used along with different mangling rules to create password guesses. Therefore in order to correctly guess a password the attacker needs to not only apply the right mangling rule but the right word also needs to be included in the dictionary. Typically, the attacker uses a dictionary that has been shown to be effective previously. In this dissertation I develop new measures to compare and analyze attack dictionaries. The experiments show that choosing the right dictionary can improve the password cracking up to 30%.

I also introduce a new method for targeted attacks. Studies show that when users change their passwords, they often slightly modify their old password instead of creating a new one. I describe how the information we have about targets can be used to help crack such passwords. Information could be names of family members, dates (such as birthday), as well as any previous passwords. I also describe an algorithm that given a set of one or more password sequences detects the differences between two or more old passwords and predicts the new password.

This research not only improves password cracking and reduces the amount of time required for cracking without any additional hardware, but also is applicable in many other areas. There have been many attempts to quantify password strength. National Institute of Standards

and Technology (NIST) publication used entropy to represent the strength of a password [2], however, researchers [3, 4] showed that the use of Shannon entropy as defined in NIST is not an effective metric for gauging password strength. I have also shown that the probabilistic context-free grammar technique can be used to create a password checker that analyzes the strength of user chosen passwords by estimating the probability of the passwords being cracked and helps users in selecting stronger passwords [5]. This system modifies the weak password slightly and suggests a new stronger password to the user. The system also dynamically updates the grammar that generally ensures the guessing entropy increases and the suggested passwords thus remain resistant to various attacks. In this dissertation I review this work and discuss new result of using this approach in estimating the strength of passwords. Our tests show that weak passwords can be distinguished from strong ones with an error rate of 1.43%. The system can also modify weak passwords to a set of strong passwords of which only 0.27% could be cracked.

The importance of this work can be seen in its applications in different areas. With the growing number of accounts users need to keep track of and with more complex password policies, users increasingly tend to store their passwords in some manner. Many users store their passwords on their computers or cellphones in plaintext. In a recent survey, it was found that 73% of users store their passwords and 34% of those save them on computers or cell phones without any encryption. In this dissertation, the problem of "identifying passwords on media" is proposed in which strings on the disk that are more likely to be passwords are identified. Automated identification could be very useful to investigators who need to recover potential passwords for further use. The problem is nontrivial because the media typically contains many strings in different locations on disk. I developed a novel approach that can successfully determine a good set of candidate strings among which the stored passwords are very likely to be found. By training on a set of revealed passwords and creating the probabilistic context-free grammar using our new patterns, we have a very good model of the way users create their passwords. This allows us to identify regular strings from passwords successfully.

In chapter 2, I explore related work and review existing techniques for password cracking. In chapter 3 I discuss our new approach of learning new patterns including keyboard and multiword; I also discuss our new metrics for comparing and improving attack dictionaries. In chapter 4, I introduce the work on targeted attack and how to create grammars that capture targeted information about the individuals. In chapter 5, I review how to use the probabilistic

context-free grammar technique to estimate the strength of passwords and I discuss the results. In chapter 6, I discuss our approach in identifying stored passwords on large disk. Finally, in chapter 7, I present conclusion and future work.

CHAPTER 2

PASSWORD CRACKING

Internet based systems such as online banking and online commerce continue to rely on passwords for authentication security. Passwords are the most common authentication technique. Human memorable passwords are thus a key element in the security of such systems. Passwords are easy to use in different domains such as social networks, emails, encryption of files and disks, and online banking protecting our sensitive data. Passwords have very convenient features such as no additional hardware to carry, ease of change, user acceptance and compatibility with encryption systems. Passwords have been important to both attackers that try to gain unauthorized access to services, and legitimate users trying to protect their clients' data or their own information. In this chapter I first give an overview of general password cracking approaches. I describe what we mean by password cracking; discuss some offline password cracking techniques, and review research work in this area. I specifically review the probabilistic password cracking technique (PPC) of [1] in some detail that is the basis for understanding the new approach proposed in this dissertation in chapter 3. I also describe the characteristics of the datasets used for training and testing throughout this dissertation. More detailed background work in each specific area is explored and presented at the beginning of each chapter.

2.1 Background and Related Work

In general, there are two types of password cracking: online and offline. In an online password cracking the system is still operational. The attacker enters a pair of username and password to the system, and the server verifies whether they match or not. A simple example of such attack is someone trying to get access to someone else's facebook account by guessing their passwords, or trying to find a pin code to unlock a cell phone. The attacker can use different tools to generate password guesses and try them on the website. The speed of online password cracking is closely related to the Internet connection speed and the target server since every guess needs to be sent over to the server. Various security features have been implemented in order to protect accounts against online attacks. Some of the common ways to prevent online password cracking include:

1. Account locking: The most common approach is to allow only a limited number of failed logins for each account. After a limited number of tries the system might lock the account and no further tries are allowed, sometimes for a period of time or sometimes other security questions are asked to prevent any unauthorized access. However, this method is vulnerable to global online dictionary attacks in which the attacker tries a password for all accounts in the system or each time for a different account to avoid account locking.
2. Delayed response: In this approach the server does not respond immediately after getting the username and password, but delays the response for a while. In this way, the attacker cannot try a large number of passwords in a reasonable time. The attacker can still try different accounts in parallel.
3. CAPTCHAS: Captchas have been used to distinguish human beings from computers. In this approach, the client is asked to enter information from a visual image with twisted words in addition to entering the username and password. Sometimes the captcha only appears after the first failed attempt. Nowadays many techniques have been developed to automatically break captchas. However, many new captcha schemes are still being proposed.

In an offline password cracking attack, on the other hand, the attacker has already obtained the password hashes or encrypted files and then tries to decrypt the file, or find the password. The attacker has already evaded the security features on the server. At this point the attacker can try different guesses at the speed his hardware supports. Since the hash functions used to store passwords are one-way functions and cannot be easily inverted, the attacker needs to repeatedly make password guesses, apply the same hash algorithm used for the target hash and compare the two hash values. If they match the password is broken, if not this process is repeated until a match is found or the attacker runs out of time. In this approach there is no limitation on the number of guesses the attacker can make to find the password, except the time he is willing to spend since the attacker is no longer limited by the system's policies and can crack the passwords at his leisure. The speed of the cracking is dependent on the resources available to the attacker. Using multiple machines or a GPU can make the cracking thousands of times faster [6]. Offline password cracking is often used as a post exploit method after an attacker has gained access to a computer or a website to retrieve more information about other users or other resources in the system. The attacker can use the cracked passwords to login to the user's

account on the same system and possibly other systems since many users reuse their passwords between different websites.

Password cracking is typically considered as an attack to gain unauthorized access to a system. However, offline password cracking can be quite useful for other purposes such as helping users recover their forgotten passwords. It is often necessary for law enforcement to crack passwords for an account, a password protected file, or to decrypt an encrypted disk in order to solve their cases. System administrators often try cracking users' passwords in order to assess the security of the system. Corporations also try offline password cracking techniques to find passwords of machines for which the password has been forgotten or for which the password is no longer available because an employee has left. Therefore, it is important to have access to the most powerful and efficient password cracking techniques. In this work we mainly focus on offline password cracking.

2.1.1 Offline Password Cracking Techniques

In offline password cracking, the attacker repeatedly makes guesses, applies the hash algorithm and compares the hash with the target hash value. Thus the most important aspect of password cracking is generating the guesses. We can categorize the most common approaches for generating guesses in offline password cracking into three main categories as follows:

(1) D*ictionary attacks*: In this approach the attacker tries a list of common words called a dictionary. The dictionary can also be used along with mangling rules that modify words and create different password guesses. A mangling rule, for example, can be appending a specific digit to the end of a dictionary word, or lower casing the dictionary word. This technique is usually fast and very popular, but attackers are still limited by the number of word-mangling rules they can apply. For example, adding a two-digit number to the end of each word in the dictionary will make the number of guesses a hundred times larger. This might delay trying other guesses that might be more useful. As can be seen, the most important challenge for most of the dictionary attacks is choosing the right mangling rules since each rule results in a large number of guesses when the input dictionary is large. The dictionary used in these types of attacks is usually a list of words that are more likely to be used by users as passwords or even passwords that have been cracked previously. Additional words from different languages can also be added

to the dictionary, particularly in cases where such linguistic information can be associated with the target of attack.

(2) Brute force attacks: In this approach the idea is to try the entire possible password space until the correct one is found. For example for a six-character password using a full key space, we can create over 697 billion combinations. While this technique is guaranteed to find the password, it is not feasible for very long passwords due to time and resource constraints. A brute force attack takes a lot of resources and a lot of time to perform so it is usually better to try this type of attack at the end of the cracking session when there is no better option. Because of this, attackers try to use more computationally efficient techniques to crack at least some of the passwords in the collection of accounts in their possession in a reasonable time.

(3) Rainbow tables: In offline password cracking the attacker makes a guess and then applies the hash function to the password guess. Often, the time consuming part is the hashing part depending on the type of hash being used. A rainbow table is a pre-computed lookup table that contains plaintext password guesses along with their hashes. In this approach, the attacker does not need to generate the guesses, but just looks up the target hash in the rainbow table. This can reduce the time of the password cracking tremendously. However, rainbow tables are not beneficial when salted hash values are used (a random data called salt is used along with the password to the hash function). The rainbow tables usually use a time-memory trade off technique known as chains to decrease the space requirements. The chain length value is determined when creating the rainbow table. When using a rainbow table with longer chain length, more hashes can be stored in the same amount of disk space, but it will make the speed of the look up slower and the possibility of the collisions higher [7]. For further information about rainbow tables please see [8]

2.1.2 Existing Password Cracking Tools

There are many existing tools available for password cracking. When choosing the right tool for password cracking many requirements need to be considered such as the platform you want to use the tool on, the capability of running it in parallel or on GPUs, whether or not it can be distributed among different systems, and also whether it is an offline or online password cracking tool. For example AirCrack [9] is a tool for WEP and WPA key cracking. It is free and open source and has been used for penetration testing. It works both on Linux and Windows.

The rainbow crack project [8] is a general-purpose implementation of Philippe Oechslin's faster time-memory trade-off technique [10]. It cracks hashes with rainbow tables. It takes a long time to pre-compute the tables but it can be hundreds of times faster than a brute force cracker once the pre-computation is finished. PasswordLastic [11] is a Windows and Office password recovery tool. THC Hydra [12], Ncrack [13] and Medusa [14] are Online Password Crackers specifically for network services and online websites.

In Medusa the brute-force testing can be performed against multiple hosts, users or passwords concurrently because of its parallel capabilities. It also has a modular design, which makes it easy to modify and add different features to it. Fgdump [15] is a newer version of the pswdump tool for extracting NTLM and LanMan password hashes from Windows. It also attempts to disable antivirus software before running. Brutus [16] is another Brute force online password cracker. It is free and only available for Windows.

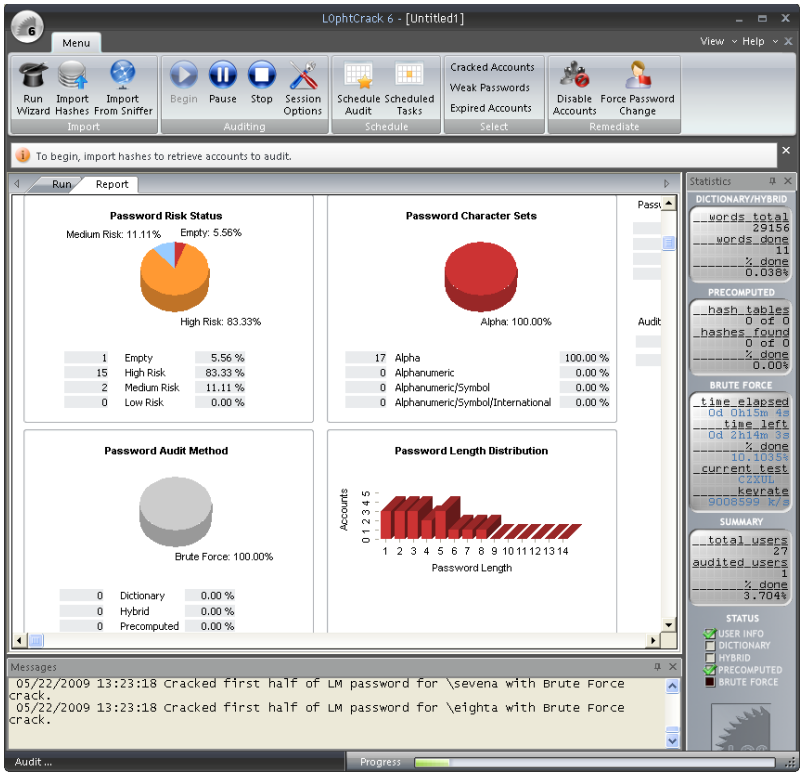


Fig. 2.1 Screenshot of L0phtcrack Password Cracker

L0phtCrack [17] cracks Windows passwords from hashes, which it can obtain (given proper access) from stand-alone Windows workstations, networked servers, primary domain

controllers, or Active Directory. It was marketed for penetration testing of system administrators. It has a very nice GUI and uses pre-computed password dictionaries, gives summary reports and supports foreign character sets for brute force attacks. A screenshot of this software can be seen in Figure 2.1.

Cain & Abel [18] is a password recovery tool for Microsoft Operating Systems. It can recover passwords by sniffing the network, cracking encrypted passwords using dictionary, brute-force and cryptanalysis attacks, recording VoIP conversations, decoding scrambled passwords, revealing password boxes, uncovering cached passwords and analyzing routing protocols. What makes Cain & Abel popular is that it is very effective in collecting passwords and password hashes from computers on local network. The downside of this program is that the password cracking tool is only effective in cracking weak passwords since its word mangling rules are limited. It has a built in support for creating rainbow tables and can use online hash databases.

ElcomSoft [19] is password recovery software, which recovers passwords protecting office documents, ZIP and RAR archives. It can perform password cracking on multiple computers, CPU cores and networked workstations to speed up the recovery. The same group has also recently released a forensic tool providing access to information stored in disks and volumes encrypted with BitLocker, PGP and TrueCrypt. The product can attack plain-text passwords protecting the encrypted containers with a range of advanced attacks including dictionary, mask and permutation attacks in addition to brute force. Although the tool supports brute force it mostly takes advantage of a variety of smart attacks that include a combination of dictionary attacks, masks and advanced permutations. The downside however is that it does not allow the attacker to use their own custom word mangling rules to use in a dictionary based attack.

AccessData [20] also has a decryption and password cracking software. It leverages graphic processing units on Microsoft Windows machines with CUDA-enabled GPUs. It can analyze multiple files at one time. It also has the ability to recover multilingual passwords. It supports a distributed network attack, which uses the power of machines across the network to decrypt passwords. DNA (Distributed Network Attack) manager coordinates the attack, assigning small portions of the key search to machines distributed in the network. It provides a

nice and easy GUI to read statistics and graphs. It also lets users add dictionaries and optimize for password attacks for specific languages. It also supports rainbow table attacks.

John the Ripper [21] is a fast password cracker for UNIX/Linux and Mac OS X. Its primary purpose is to detect weak Unix passwords, though it supports hashes for many other platforms as well. There is an official free and open source version and a community-enhanced version (with many contributed patches). It has many hash functions built into it and it has the ability to accept guesses from an external program piped into it. Therefore, it is easy to use the hash functions implemented by John the Ripper for different password sets and for testing. It is also possible to export guesses generated by it to other programs in order to get statistics for testing and comparing purposes. John the Ripper is one of the most recognized password crackers, is easy to use and has a community that keeps adding on patches and more capabilities to it. Screenshot of John the Ripper options are shown in Figure 2.2.

```
Usage: john [OPTIONS] [PASSWORD-FILES]
--config=FILE             use FILE instead of john.conf or john.ini
--single[=SECTION]        "single crack" mode
--wordlist[=FILE] --stdin wordlist mode, read words from FILE or stdin
                  --pipe  like --stdin, but bulk reads, and allows rules
--loopback[=FILE]         like --wordlist, but fetch words from a .pot file
--dupe-suppression        suppress all dupes in wordlist (and force preload)
--encoding=NAME           input data is non-ascii (eg. UTF-8, ISO-8859-1).
                          For a full list of NAME use --list=encodings
--rules[=SECTION]         enable word mangling rules for wordlist modes
--incremental[=MODE]      "incremental" mode [using section MODE]
--markov[=OPTIONS]        "Markov" mode (see doc/MARKOV)
--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. It will be overwritten
--show[=LEFT]             show cracked passwords [if =LEFT, then uncracked]
--test[=TIME]             run tests and benchmarks for TIME seconds each
--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[:MAX]    load salts with[out] COUNT [to MAX] hashes
--pot=NAME                pot file to use
--format=NAME             force hash type NAME: afs bf bfegg bsdi crc32 des
                          dmd5 dominosec dragonfly3-32 dragonfly3-64
                          dragonfly4-32 dragonfly4-64 drupal7 dummy dynamic_n
                          epi episerver gost hdaa hmac-md5 hmac-sha1
                          hmac-sha224 hmac-sha256 hmac-sha384 hmac-sha512
                          hmailserver ipb2 keepass keychain krb4 krb5 lm lotus5
                          md4-gen md5 md5ns mediawiki mscash mscash2 mschapv2
                          mskrb5 mssql mssql05 mysql mysql-sha1 nethalflm netlm
                          netlmv2 netntlm netntlmv2 nsldap nt nt2 odf office
                          oracle oracle11 osc pdf phpass phps pix-md5 pkzip po
                          pwsafe racf rar raw-md4 raw-md5 raw-md5u raw-sha
                          raw-sha1 raw-sha1-linkedin raw-sha1-ng raw-sha224
                          raw-sha256 raw-sha384 raw-sha512 salted-sha1 sapb
                          sapg sha1-gen sha256crypt sha512crypt sip ssh
                          sybasease trip vnc wbb3 wpapsk xsha xsha512 zip
--list=WHAT               list capabilities, see --list=help or doc/OPTIONS
--save-memory=LEVEL       enable memory saving, at LEVEL 1..3
--mem-file-size=SIZE      size threshold for wordlist preload (default 5 MB)
--nolog                   disables creation and writing to john.log file
--crack-status            emit a status line whenever a password is cracked
--max-run-time=N          gracefully exit after this many seconds
--regen-lost-salts=N      regenerate lost salts (see doc/OPTIONS)
```

Fig. 2.2 Screenshot of John the Ripper v1.7.9-jumbo-7 Options

Hashcat [22] is a command line interface application designed to take advantage of multiple cores of modern CPUs for faster password cracking. Hashcat works with both CPU and GPUs and supports multi-threading. It supports more than 40 different hash types and many attack modes such as dictionary attacks, rule-based attacks, table lookups, and brute force.

The main difference between the existing tools is not the technique they are using for generating the guesses, but the different types of hashing algorithms that they support or the operating systems they work on. Almost all of these password crackers use the brute force technique or use dictionary attacks with mangling rules. Some can combine both and make brute force faster with some improvements. There wasn't much work or improvement in the algorithm used for password guessing until the probabilistic password cracking work (PPC) [1] in which a context-free grammar is constructed by training on real user passwords. The grammar in turn is used to generate guesses in highest probability order. We review this work next. We later compare the result of our improved password cracker with PPC and two of the most popular password crackers (John the Ripper and Hashcat) described above.

2.1.3 Probabilistic Password Cracking (PPC)

The explanations in this section are drawn from Weir et al. [1] where the authors used probabilistic context-free grammars to model the derivation of real user passwords and the way users create their passwords. The basic idea of this work is that not all the password guesses have the same probability of being the target password. For example, appending the number "2015" to a password guess might be more probable than appending a random number "6710" at the end of a dictionary word since users are more likely to use dates and year in their passwords. The main idea is then to generate guesses in a decreasing order of probability.

A context-free grammar is 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 $\alpha \rightarrow \beta$, where α is a single variable and β is a string consisting of variables or terminals. The set of all strings derivable from the start symbol is the language of the grammar. Probabilistic context-free grammars [23] have probabilities associated with each production such that for a specific left-hand side variable, all the associated production probabilities add up to 1. In Weir et al. [1], strings consisting of alphabet symbols are denoted as L, digits as D, special characters as S and Capitalization as C. The authors also associate a

number to show the length of the substring. For example, the password "football123!$" would be $L_8D_3S_2$. Such strings are called the base structures. There are two steps for this technique: the first is constructing the probabilistic context-free grammar from a training set of publicly disclosed real user passwords, and the second is generating the actual guesses in decreasing probabilistic order using the context-free grammar.

2.1.3.1 Training. The first step is to automatically derive all the observed base structures from the training set of passwords and their frequency of appearance in the training set. Then the same information for the probability of the digits, special characters and capitalization will be obtained from the training set.

The probability of any string derived from the start symbol is then the product of the probabilities of the productions used in its derivation. Table 2.1 shows a very simple example of a probabilistic context-free grammar.

Table 2.1 Example of a Probabilistic Context-Free Grammar

Left Hand Side	Right Hand Side	Probability
S→	$D_3L_3S_1$	0.8
S→	S_2L_2	0.2
D_3→	123	0.76
D_3→	987	0.24
S_1→	!	0.52
S_1→	#	0.48
S_2→	**	0.62
S_2→	!@	0.21
S_2→	!!	0.17
L_3→	dog	0.5
L_3→	cat	0.5

Using this grammar, for example, we can derive password "987dog!" with probability 0.04992. Note that in this work the words replacing the L part of base structures come from the dictionary with probabilities equal to one over the number of words of length i.

S → D3L3S$_1$ → 987L$_3$S$_1$→987dogS$_1$→987dog!

$0.8 \times 0.24 \times 0.5 \times 0.52 = 0.04992$

2.1.3.2 Probability Smoothing. No matter how large the training set, it will not include all possible values for digits, special symbols and base structures. There are many numbers and combinations of the special characters that may not occur in the training set. If a specific number is not included in the grammar, it will not be used when generating guesses. Ideally, a good password cracker needs to try all possible values. In order to solve this problem, the authors [7] added the values that were not found in the training set to the context-free grammar with lower probability values. Let $\boldsymbol{C}$ be number of different categories, and $\boldsymbol{N_i}$ be the number of items found in $\boldsymbol{i}$th category. They used a variant of Laplace smoothing where the probability of each element $\boldsymbol{i}$ is as follows:

$$\boldsymbol{p_i} = \frac{\boldsymbol{N_i}+\boldsymbol{\alpha}}{\sum \boldsymbol{N_i}+\boldsymbol{C\alpha}} \tag{2.1}$$

where α is between 0 and 1. This has been implemented only for digits and special symbols below a certain length.

As an example, if we consider smoothing the probabilities of digits of length two, *C=100* since there are 100 different numbers of length 2. Suppose *α = 0.1* and we have found "11" 8 times, "99" 12 times, and "22" 30 times. The probability of all the other digits that are not found in the training set can be calculated as follows:

$$p_i = \frac{0 + 0.1}{(8 + 12 + 30) + 100 \times 0.1} = \frac{0.1}{60} = 0.0016$$

2.1.3.3 Guess Generating. After obtaining the probabilistic context-free grammar by training on a set of real user passwords, the guess generator generates password guesses in a decreasing order of the probability using the context-free grammar obtained from the previous step. It uses an attack dictionary to replace the alpha strings in base structures. It can take more than one dictionary with different probabilities associated to each. While it is not hard to generate the most probable password guess (you just need to replace all the base structures with the highest probability pre-terminals and then selects the pre-terminal with the highest probability), generating the next password guesses is not trivial. The authors have developed an algorithm called "Deadbeat Dad" which uses a priority queue and is also efficient in terms of memory usage. As each pre-terminal is popped from the queue and password guesses related to that pre-terminal are generated, the function determines which children of that node have to be pushed into the priority queue [7].

Their experiments show that using a probabilistic context free grammar to create the word-mangling rules through training over known password sets can be a good approach. It allows us to quickly create a rule set to generate password guesses for use in cracking unknown passwords. When compared against the default rule set used in John the Ripper, this method outperformed it by cracking 28% - 129% more passwords, given the same number of guesses. Since this password cracker is one the most recent techniques and has been shown to be very effective in cracking passwords, we built upon this system to capture more patterns in passwords. I was fortunate to have access to all the relevant code, and was able to use the system as a basis for my work to implement the new password cracker, discussed in chapter 3.

2.1.4 Recent Password Cracking Techniques

The most well-known password cracking algorithms are those that are based on Markov models [24] and the probabilistic context-free grammar [1]. In this dissertation we mainly focus on the probabilistic context-free grammar approach of Weir et al. [1] and compare our results against this work since it is often cited by many authors [25, 26] as the state-of-the-art in password cracking. In this section, however, we discuss two of the most recent studies in password cracking that claim to perform better than the original PPC [1]. Later in chapter 3, the result of our new password cracker is compared against these two approaches.

Ma et al. [25] explored different probabilistic models for password cracking extensively. The authors conducted studies on different Markov models with different normalization techniques and compared it against original PPC [1] and found that the whole-string Markov model outperforms PPC. Whole string Markov model has been used in John the Ripper password cracker [21] and the adaptive password strength meter [27]. In an *n*-gram Markov model the probability of each character is conditioned on the probability of the *n* characters that come before it. The probability of each string is then calculated by multiplying the conditioned probabilities. The work by Ma et al. [25] does not generate guesses in highest probability order. The authors only estimate the probability of passwords and in order to show the effectiveness of the cracker they generate probability-threshold graphs in which they only calculate the probability of each password to see where it would have appeared if they were trying to crack it.

Veras et al. [28] created a system to semantically classify passwords and generate guesses by combining the probabilistic context-free grammar approach [1] with the natural language

processing technique. In this section we briefly review their work. Later in chapter 3 I show the result of comparing our work and discuss how their approach is different than ours.

Veras et al. [28] focus on classifying words by their semantic content. The authors first generate all possible segmentations of a password and then determine the ones with the highest coverage using a source corpus. N-gram probabilities from a reference corpus are used to select the most probable segmentation with a back-off approach. Once the words are broken down, the authors tag each word with its part of speech using NLTK. The authors use WordNet to classify the words into semantic categories and use these categories to develop a context-free grammar. While this work has some drawbacks as explained later, it can be a useful guide in our future work applying smoothing to multiwords.

2.2 Characteristics of Datasets

Throughout this research study we use real user password lists that have been publicly disclosed as a result of hacking attacks. Hackers usually post the obtained password sets to forums or on compromised web servers. Some of the datasets have been captured as plaintext as a result of a phishing attack or because the webserver stored passwords in plaintext. Unfortunately, not all passwords in these datasets represent real passwords. For example, in some phishing attacks, some users recognized the phishing site and entered irrelevant data. Other datasets have been obtained as hash sets and have been broken by hackers and password communities.

We test the effectiveness of our system throughout this work using several different datasets of revealed passwords. In our tests, we always randomly select a number of the passwords as our training and test sets. We ensure that the test set is always different from our training set. In this section, we describe the revealed password sets, and how we create our datasets using these password sets. We also explore some characteristics and statistics of the datasets referenced throughout this dissertation.

(1) Yahoo Set: This set is one of the most recent plaintext password sets that has been leaked by hackers in 2012. It contains about 453,000 login credentials. The data appears to have originated from Yahoo Voices platform and is a result of an SQL injection attack [29]. We have randomly chosen 300,000 passwords from this set as our training set and we call this set Yahoo-training. The remaining 142,762 passwords create our test set (Yahoo-test).

(2) Combined Set: Our goal in general in our tests was to have reasonable and similar sized test sets wherever possible. We also wanted to use as many revealed password sets as we could for training. In this set, we used a mix of passwords from different revealed sets. Combined-training contains ½ million from Rockyou [30], combined with 30,998 MySpace [31] and 4,874 from Hotmail [32] passwords. Combined-test contains the same number of passwords from the original lists as in Combined-training. Note that these two sets do not overlap and contain different passwords.

The Rockyou list [30] contains over 32 million plaintext passwords and is a result of an SQL injection attack in 2010 on Rockyou.com, which made applications for social networking websites such as Facebook and MySpace. The MySpace list is the result of a phishing attack on MySpace.com. The list contains about 62 thousands plaintext passwords and was publish on October 2006. The Hotmail list has been obtained in October 2009 and contains about 10 thousands passwords [32]. Since the Hotmail and MySpace passwords lists are fairly small, we have combined them with Rockyou password set.

(3) CSDN Set: The CSDN set is the result of an attack on csdn.net, a Chinese language Software Developer Network forum in 2011 and it contains about 6 million passwords [33]. There were a few passwords that contained Chinese characters, which we have removed from this set. We then created CSDN-training set with 300,000 passwords randomly chosen from this set. The CSDN-test set contains 150,000 passwords. It is shown in Table 2.2 that compared to other password sets, CSDN set has fewer passwords with lengths less than 8. As seen in Table 2.3, CSDN also has more passwords containing digits than other password sets.

In the password cracking tests throughout this dissertation we use *dict-0294* [34] as the primary attack dictionary and *common_passwords* [21] as the secondary dictionary with 0.05 and 0.5 probability values respectively, unless stated otherwise. We created a training dictionary by augmenting the EOWL [35] list by common proper names [36] and top words from television and movie scripts [37]. EOWL was originally designed for Scrabble style word games.

Table 2.2 shows the password length distributions for sets used in this dissertation. In Yahoo set, passwords with lengths between 7 and 11 represent about 87% of all passwords. Passwords with lengths between 6 and 10 contains about 96.6% of all passwords in Combined set, and in CSDN set, passwords with lengths between 8 and 12 cover about 90.5% of all passwords. Table 2.3 shows the statistics of containing digits, alpha strings, special symbols,

keyboard and multiword (as discussed in chapter 3) for the above password sets. It shows that data sets are similar in having higher percentages of passwords containing digits and lower case alpha strings, and fewer passwords containing upper case alpha strings and special symbols. However, CSDN has more passwords containing digits and fewer passwords containing alpha strings and multiword than other password sets. In fact CSDN has an unusually large number of passwords consisting of only digits (3 times as often as Combined set).

Table 2.2 Password Length Information

Length	Yahoo test (%)	Yahoo training (%)	Combined test (%)	Combined training (%)	CSDN test (%)	CSDN training (%)
2	0.03	0.02	0.0046	0.003	0.007	0.004
3	0.02	0.01	0.023	0.02	0.013	0.014
4	0.06	0.07	0.22	0.21	0.09	0.13
5	0.59	0.63	3.96	3.94	0.54	0.53
6	1.2	1.2	25.63	25.76	1.28	1.32
7	18.13	17.91	19.36	19.32	0.28	0.27
8	14.79	14.83	20.12	20.02	36.45	36.36
9	26.9	26.91	12.28	12.31	24.09	24.17
10	14.93	14.88	9.21	9.27	14.43	14.46
11	12.31	12.4	3.59	3.58	9.83	9.72
12	4.76	4.81	2.09	2.09	5.74	5.71
13	4.86	4.92	1.31	1.28	2.59	2.64
14	0.6	0.6	0.86	0.85	2.42	2.44
15	0.33	0.34	0.54	0.55	1.16	1.17
16	0.19	0.19	0.41	0.39	0.77	0.76
17-30	0.3	0.28	0.39	0.41	0.31	0.3

Table 2.3 Password Characters Information

	Yahoo test (%)	Yahoo training(%)	Combined test(%)	Combined training(%)	CSDN test(%)	CSDN training(%)
Contains digits	64.64	64.78	55.14	55.08	86.95	87.14
Contains lower	92.85	92.82	80.98	80.98	51.43	51.35
Contains upper	8.49	8.51	6.03	5.97	4.65	4.61
Contains symbol	2.86	2.83	4.05	4.02	3.67	3.62
Contains keyboard	6.67	6.47	5.3	5.29	7.56	7.43
Contains multiword	26.77	26.86	26.97	26.82	11.43	11.28

CHAPTER 3

NEXT GENERATION PROBABILISTIC PASSWORD CRACKER

While the probabilistic password cracker of Weir et al. [1] was very successful at the time, there was not much improvement in password crackers following this work. The probabilistic password cracker although performs much better than other rule-based password crackers, but considers fairly simple components, containing only alpha strings, digits and special characters to represent user passwords. In practice, users with more knowledge about security create more complex passwords and a password cracker needs to be adapted to these changes. Passwords containing keyboard patterns, Leetspeaks (replacing characters and digits for alphabets in a word such as: P@ssword), multi-words and phrases that do not exist in common password cracking dictionaries, are examples of such techniques that is generally not addressed directly by password crackers. In most cases, the approaches simply add the common keyboard combinations and common words with leetspeaks into the dictionary. In this chapter we discuss learning new patterns that are more likely to appear in real user passwords as an extension to the probabilistic password cracker of Weir et al. [1]. By assigning probabilities to these new patterns through the use of probabilistic context-free grammar, we can capture both appropriate words and fine-grained word mangling rules, in a unified framework. Furthermore, this enables us to keep generating guesses in highest probability order, which is the optimal attack. In this chapter I describe the work that I have done on extending patterns for probabilistic password cracking and developing new metrics for improving attack dictionaries. The work in this chapter has been accepted for publication [38].

3.1 Keyboard Combinations

The goal of this section is to understand how users use keyboard patterns and how it can be incorporated into the probabilistic password cracker. A keyboard pattern is a sequence of keystrokes that are made on the keyboard without paying attention to the actual characters and their relation to each other except their closeness on the keyboard. This closeness helps users remember keyboard combination passwords better. We define a keyboard pattern as a sequence of at least three contiguous characters starting from some particular key. Contiguous characters

are keys that are physically next to a certain key or it can be the same character repeated. For example in the keyboard shown in Figure 3.1 contiguous keys for character **j** can be: **u** (upper left), **i** (upper right), **h** (left), **j** (same), **k** (right), **n** (lower left), **m** (lower right). A typical example of a keyboard pattern used as a password is "qwerty". This pattern can be also combined with other components to create the password, for example "qwerty952!".

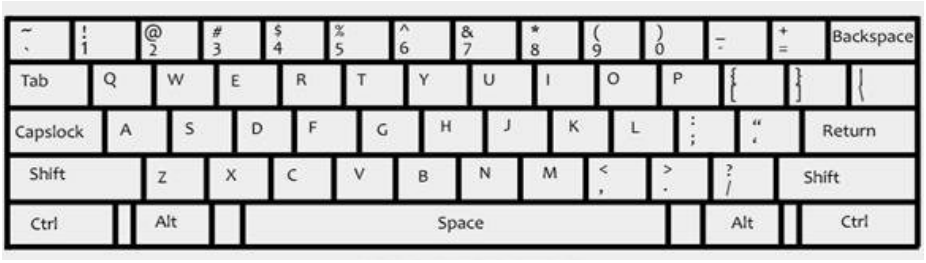


Fig. 3.1 Example Keyboard

Typical password crackers incorporate such patterns by adding them in the attack dictionary. However, dictionaries do not differentiate patterns by their probability of occurrences and one can only afford to add a limited number of such patterns. We instead solve the problem by modifying the probabilistic context-free grammar. Using this approach the incorporation of the keyboard patterns becomes automatic during training and cracking. By smoothing the keyboard probabilities (described in section 3.1.3), we can automatically generate new patterns that have not been seen in the training set. There are not many studies that explore the keyboard patterns, their strength and how often they are used as passwords. De Luca et al. [39] have studied PINs used for authentication and have learned that users create an overlaying shape and memorize the geometrical figure instead of memorizing the actual numbers. The authors introduced PassShape, an authentication method that uses the shapes without the numbers, which is easier to remember for users. Schweitzer et al. [40] describe a way to pictorially illustrate a shape on the keyboard. They connected sequentially pressed keys with an arc. To visualize a key that has been pressed multiple times, they create concentric petals. In a small experiment, they gave 161 users a brief tutorial on how to create patterns, and they gathered 250 unique patterns. They then generated a number of keyboard patterns from the most common shapes found, and added these to a dictionary for use in a standard attack (a common way of using keyboard patterns). For their testing, they obtained 11 passwords from their institution and used the

dictionary for password cracking. They were able to crack 2 of the passwords while John the Ripper was not able to crack any. While their work in regard to visualizing the pattern and identifying the most common patterns was interesting, the attack shown by this work is not different from previous typical keyboard attacks. In the next section I discuss how we identify keyboard patterns in the training data, and how to use this information in the cracking phase.

3.1.1 Finding Keyboard Patterns and Ambiguity Issues

As mentioned previously, a keyboard pattern is modeled as a sequence of at least three contiguous characters on the keyboard. We allow both upper case and lower case characters. The algorithm looks for the longest keyboard pattern in a password without being concerned about what type of characters they are. We use the symbol K to represent a keyboard component in the grammar. For example, given the password "qw34%rt952", the original probabilistic password cracker would have parsed this password to $L_2D_2S_1L_2D_3$, while our new algorithm considers this as K_7D_3 (qw34%rt: keyboard pattern of length 7 and 952: digit of length 3). It is also possible to find more than one keyboard pattern in a password. For example, "qwerty521qazxsw" is parsed as $K_6D_3K_6$.

When identifying keyboard patterns in passwords, we also capture the *shape* they create on the keyboard. In order to keep track of this information, we use the following notation and symbols: an *upper left* key relative to the current key is represented by the symbol **u**, an *upper right* key is denoted by **v**, the *same* key is denoted by **c**, the *left* key is denoted by **l**, the *right* key is represented by **r**, the *lower left* key is denoted by **d** and the *lower right* key is denote by **e**. For example, given the password "qw34%rt952", the keyboard pattern (qw34%rt) of length 7 starts with q and has the keyboard shape ***rvrrdr***.

The original probabilistic context-free grammar introduced in [1] is unambiguous. It is easy to see the unambiguity since the simple base structures introduced in this work (L, D, and S) are mutually exclusive. In our new approach, when incorporating the keyboard patterns we face the ambiguity problem since a K-structure could contain multiple character sets. An ambiguous grammar is a grammar for which there are more than one derivation trees that correspond to a terminal string. Consider the following simple example shown in Figure 3.2. Using the natural language grammar we can generate a string with two derivation trees. Note that

both trees generate the same terminal sentence (string). Although the terminal is the same, there are two different grammar productions (and two different meanings) for this terminal [41].

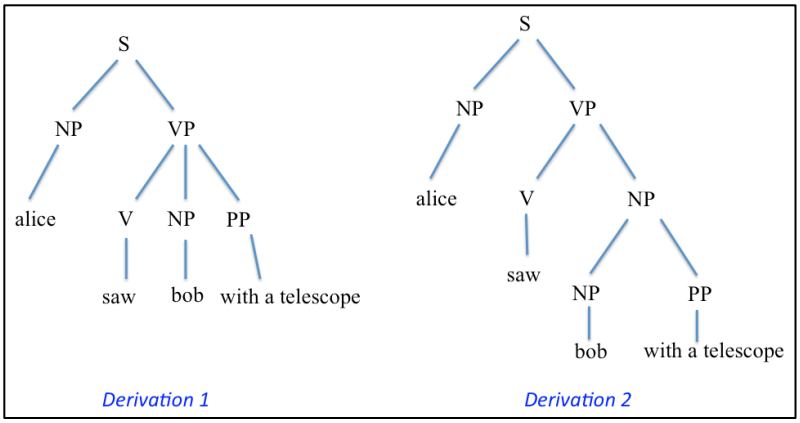


Fig. 3.2 Derivation Trees with an Ambiguous Natural Language Grammar

When adding keyboard structures to the grammar without any special considerations, we could also face the ambiguity problem. Consider the following simple grammar as an example. This grammar is clearly ambiguous since the string "cat1234" can be generated with two different derivation trees as shown in Figure 3.3.

$S \rightarrow L_3D_4 \mid L_3K_4$
$L_3 \rightarrow$ cat | dog
$D_4 \rightarrow$ 1234 | 9637
$K_4 \rightarrow$ qwer | 1234

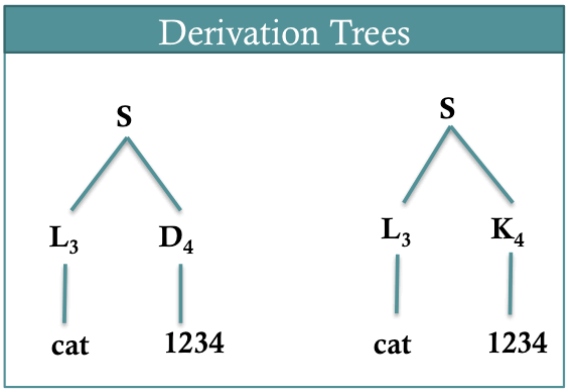


Fig. 3.3 Derivation Trees for an Ambiguous Grammar using K-Structures

In password cracking, different derivations for the same guess means that the same password will be guessed multiple times, each with its own probability value. This obviously reduces the efficiency of the guesser. One might wonder why we do not want to allow ambiguous grammars as part of the probabilistic password cracker. Aside from generating duplicate guesses, the probability values of such password guesses are incorrectly calculated. The correct probability value should be the sum of all possible derivations. Prescher [41] shows that there are algorithms such as expectations maximization algorithm (EM) that can produce probabilities for such grammars by training on a set of derivation trees. However, this approach cannot be used for our purpose since we do not have access to such data (derivation trees of passwords and their frequencies). In other words, when we come across a password like cat1234, there is no obvious way for us to determine whether the user meant "1234" as a keyboard pattern or as a number. Also, generating the guesses in highest probability order from the grammar relies on the grammar being unambiguous. This ensures that there is well-defined probability for each guess that only depends on a single unique derivation.

We therefore try to maintain an unambiguous context-free grammar by limiting the terminals each base structure can derive. In the ambiguity that arises from situations like the above example where we have keyboard patterns consisting of pure digits or special characters, we decided to preferentially consider them as digit (or special symbol) components rather than keyboard components. Prior to this decision, we looked at several data sets and picked out base structures that have keyboard patterns that contain only digits. (These are the patterns that can be interpreted as both keyboard and digit components.) We then tried to find such structures in another data set. The results showed that approximately 70% of the times these patterns appeared to be digits; thus by treating them as digit components we would have a better chance of guessing related passwords. We found similar results for S components.

In summary, the following rules are defined to determine whether or not a particular substring in a password should be classified as a keyboard pattern (K-structure) rather than an original structure (L, D, S):

1. If a substructure contains only digits or only special symbols, we classify it as a D or S component.

2. Longest keyboard patterns of at least 3 characters length that does not fall under the first rule is classified as a K component. For example *e4e458* would be K_5D_1 as the maximal length keyboard substring must be used.

Although these rules can avoid the ambiguity problem in a vast majority of cases, but in rare cases we will still be generating duplicate guesses. For example, both base structures L_4D_2 and K_6 would be able to generate password guess "were45". We will discuss these examples in more details in section 3.1.2.

Table 3.1 shows examples of the original base structure compared to the keyboard base structure. Similar to the other components that are found in the training set, keyboard combinations will be stored in the grammar as a pair of the actual pattern found along with its probability value. While we determine the patterns and the base structures, we also count their frequencies. These frequencies are ultimately turned into transition probabilities. We also determine the counts of components such as those in D_1, ... D_j for j the largest D-structure and similarly for components in S_1, ... S_m for m the largest S-structure (special symbol). For example, encountering the password "asd1234qw" would increase a D_4 count for substring 1234 by one. We also increase the counts of the keyboard components in K_3, ... K_p for p the longest keyboard pattern.

Table 3.1 Keyboard Base Structures during Training

Password	Original Base Structure	Keyboard Base Structure
qwerty	L_6	K_6
R5T6	$L_1D_1L_1D_1$	K_4
!@#$	S_4	S_4
tyu54uyt	$L_3D_2L_3$	$K_3D_2K_3$
E3$4	$L_1D_1S_1D_1$	K_4
4567	D_4	D_4

The K-structure can be handled the same way as D and S-structures when generating guesses. Since we have preserved all aspects of a context-free grammar, we are able to automatically generate password guesses in highest probability order with keyboard combinations appearing in their appropriate probabilistic order.

3.1.2 Using a Training Dictionary

With the above rules for identifying keyboard combinations, a password such as "ease12" would be classified as $L_1K_3D_2$. Our initial assumption was that most probably the user that selected this password did not mean to have one alpha character followed by a keyboard combination, and instead the user meant the word 'ease' followed by two digits. We hypothesized that it might be preferable to view such components as English words followed by digits (L_4D_2) rather than a keyboard component. In order to eliminate such spurious keyboard combinations we analyzed the alpha sequence of the passwords more carefully. We introduce the idea of using a training dictionary. During the training phase of determining the base structures we analyze sections of passwords that could be forming a keyboard pattern but are also alphabet letters that could be a word in the training dictionary. Each alpha sequence is looked up in the training dictionary to recognize any English word. If it is in the training dictionary it will be classified as L, if not it will be considered for further evaluation as part of a keyboard pattern.

The training dictionary is used to resolve ambiguity in this case as well as in more complex situations such as determining multi-words discussed in section 3.2. Note that the *training dictionary* is different from the *attack dictionary*. The attack dictionary contains words or parts of the common words in passwords, sometimes combinations of letters with no meaning (that appeared to be useful in password cracking combined with mangling rules), abbreviations and other common phrases on the Internet or the relevant website. The attack dictionary is used to replace the alpha string component (L) of the base structure when generating guesses.

On the other side, the training dictionary contains actual English words and common proper names. The training dictionary can be very large and since the training phase can be done before the actual cracking session starts, it does not affect the cracking time. However, the size and the actual words contained in the attack dictionary can affect the efficiency of the password cracking session. Every word in the attack dictionary will be tried with different mangling rules and a very large attack dictionary can make the cracking time unnecessarily long and not efficient. If the words in the dictionary are not useful words for password cracking, the cracking session can take days without a single password being cracked. Attack dictionaries are discussed in more details in section 3.3.

3.1.3 Probability Smoothing for Keyboard Patterns

As mentioned in section 2.1.3.2**,** the original context-free grammar of Weir [1] is capable of smoothing D and S structures by using Laplace smoothing. Consider for example the component D_2. It is possible that some of the two digit values are not found during the training. In this case, no probability will be assigned to these values and they will not be used in generating guesses. Probability smoothing helps in assigning lower probability values to values that are not found during training. Smoothing keyboard patterns though is not straightforward since it is not easy to find the complement of the found ones. In this section, I discuss our approach for smoothing keyboard combinations.

Recall that during the training process we capture data about keyboard shapes (such as *rrrrr* for "qwerty") found in the various keyboard patterns. Table 3.2 shows a small sample of keyboard patterns of length 6, and keyboard shapes of length 5 that were found during a training session.

Table 3.2 Keyboard Shapes and Patterns

Shapes	Probability
rrrrr	0.520
eveve	0.102
eeruu	0.058
llllll	0.036
rdrdr	0.024
rlrlr	0.020
vdvdv	0.007
rrrrd	0.007

Patterns	Probability
qwerty	0.488
2w3e4r	0.093
qazxsw	0.056
zxcvbn	0.025
qwaszx	0.018
poiuyt	0.012
12345r	0.007
ytrewq	0.007

Smoothing keyboard patterns allows new keyboard combinations to be generated while guessing. Smoothing every possible keyboard combination would result in a very large number of guesses that might not even be useful. We therefore decided to only consider more common shapes found in the training and smooth keyboard patterns based on found shapes. We view a smoothed element as a specific keyboard shape applied to an applicable starting character. For example, the keyboard shape *rrrrr* allows us to start from every character on the keyboard and

create a keyboard combination using this shape. Thus, for keyboard shapes that we find in the training set, we smooth over all starting characters excluding those that are not feasible (starting at *m* for shape *rrrrr*). This approach is a reasonable compromise between smoothing everything and not smoothing anything. Essentially, instead of smoothing across all keyboard strings, we smooth across each keyboard shape found of a specific length. The smoothing function giving the probability of a keyboard pattern ***p*** of shape ***s*** is:

$$prob(p) = prob(s)\frac{N_i + \alpha}{\sum N_i + C\alpha} \tag{3.1}$$

where:

Prob(s) is the probability of the keyboard shape ***s*** given the length of the keyboard pattern

N_i is the number of times the *i*th keyboard pattern (of this shape) was found

α is the smoothing value between 0 and 1

ΣN_i: the sum of counts of the patterns found for shape ***s***

C is the total number of unique patterns for this shape

3.1.4 Testing and Result

In order to see the effectiveness of the new keyboard grammar in comparison to the initial grammar we ran several different tests. We used different revealed password sets that are commonly used by researchers. We created test and training sets of different sizes and origins for our experiments. In these series of tests, we consider *keyboard* alone, and *Keyboard plus Dictionary* in which we use a training dictionary to separate words from keyboard patterns. We also consider two variations for each of these based on whether we smooth or not.

In our first set of tests, we trained on Combined-training and the target set was Combined-test. The datasets and dictionaries are described in section 2.2. Figure 3.4 shows the results of comparing each of the 4 variations across portions of the cracking curve (early, middle and late). By cracking curve we mean graphing the percentage of passwords cracked on the Y-axis against the number of guesses generated. In these series of tests we generated about 85 billion guesses. We also use the notion of improvement to compare two different cracking curves U(x) and V(x). By definition, the *improvement* of U over V at x is simply (U(x) - V(x)) / V(x).

The goal of using a training dictionary was to maintain a structure that is probably a word for the user as an L-structure and thus try a variety of words eventually as replacements in that structure. We wanted to distinguish keyboard patterns that are really, in some sense, user keyboard patterns and are not "artifact" keyboard patterns simply because they have serendipitously an embedded keyboard structure (such as the word **ease**). Although eliminating such keyboard patterns seemed natural and we expected that this kind of grammar would perform better, the results show otherwise.

Figure 3.4 shows that Smoothed Keyboard grammar is not as effective as the other grammars in the very beginning (Early), but it is clearly the best very soon (about 1.5 billion guesses) and maintains this for the rest of the cracking curve. In general both Smoothed Keyboard and Keyboard are clearly better than the Dictionary versions with Smoothed Keyboard being slightly better.

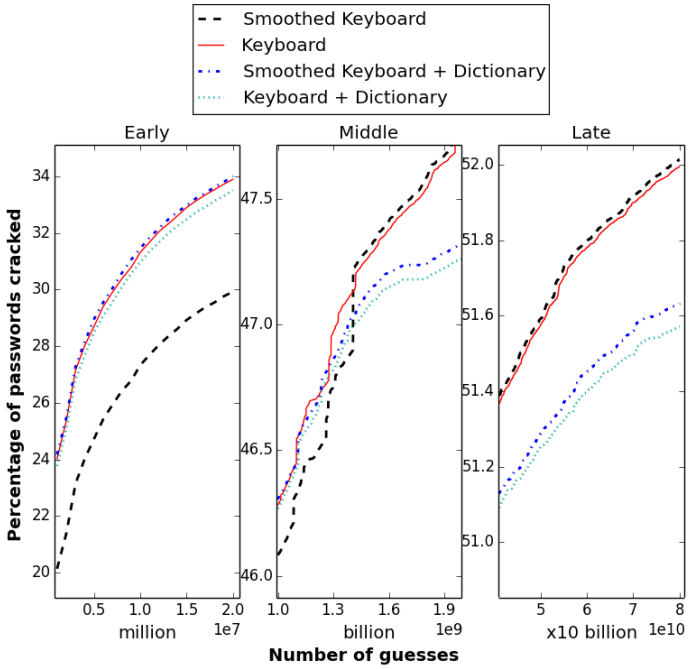


Fig. 3.4 Results for Keyboard Versions using Combined-set for Early (up to 20 million), Middle (1-2 billion), and Late (40-85 billion)

In Figure 3.5 we compare Smoothed Keyboard grammar with John the Ripper and original PPC. As can be seen, Smoothed Keyboard is more effective than PPC in most of the cracking session, and is also much more effective than John the Ripper over a major part of the

password cracking curve. After about 34 billion guesses John the Ripper overtakes the original grammar while it still cannot outdo Smoothed Keyboard grammar until after about 52 billion guesses. We also repeated the same test using Yahoo and CSDN sets as described in section 2.2. In both cases, Smoothed Keyboard grammar is better than original PPC, which in turn is better than John the Ripper over the whole cracking curve.

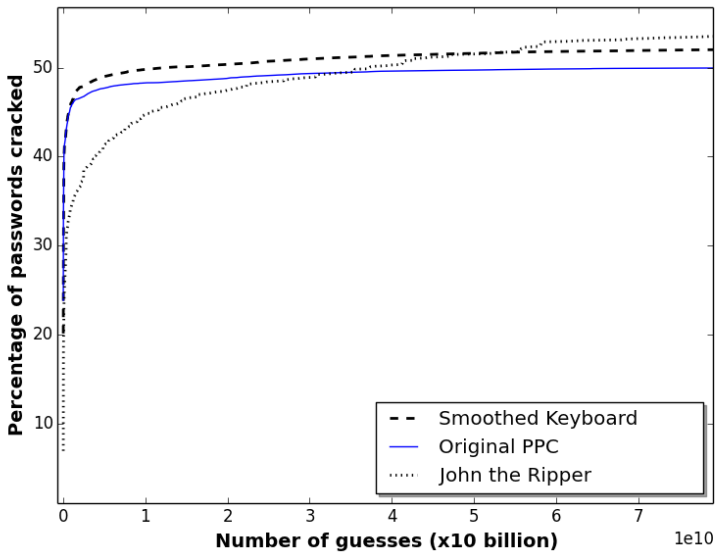


Fig. 3.5 Comparing Password Crackers using Combined-set

3.2 Enhancing Identification of Alpha Strings

Many times when creating long passwords people prefer to use longer words, a sentence or a phrase in the alpha part of the password. The original probabilistic password cracker considers the sequence of alpha characters as an L structure and in the cracking session it replaces the L structure with a word of that length from the attack dictionary. For example, the cracking module looks for a word of length 15 in the dictionary when a base structure containing L_{15} is reached. Most probably there does not exist many words of this length in the dictionary and if any, not all combinations of phrases and multi-words exist.

Our goal is to better understand alpha strings or L-structures. Most common examples are multiple occurrences of a word within alpha strings such as *johnjohnjohn12*, or passphrases such as *goodboy* or *iloveyou*. Although some of the popular combinations might be in the attack dictionary, it is not easy to add all possible combinations to the dictionary. In this section we

discuss how the alpha string patterns are modeled in our system and how we use this information in guessing.

3.2.1 Detecting Alpha String Patterns

In order to support detection of relevant patterns in alpha strings, a training dictionary is used as previously defined in section 3.1.2. During training, we learn the following different categories of L-structures. Table 3.3 shows examples of each category and the frequencies of each category in one of our sample sets.

- A-word: A single word found in the dictionary
- R-word: A word in the dictionary, repeated once
- R-pattern: A non-dictionary word, repeated once
- M-word: Two or more consecutive dictionary words excluding R-words
- A-pattern: Alpha string not in any previous category

Table 3.3 Classifications of Alpha Strings

Category	Frequency in a sample test set	Example
A-word (a single word)	44%	password
R-word (repeated word)	0.98%	boatboat
R-pattern (repeated pattern)	0.35%	xyzxyz
M-word (multiword)	40.4%	iloveyou
A-pattern (other pattern)	14.2%	ahskdi

In order to detect A-words in the training set, we simply check if the L-structure is a word in the training dictionary. For R-patterns we first check for a repetition and then if the pattern is in the dictionary we categorize is as R-word. If the L-structure is neither of these two categories, we apply our M-word algorithm to distinguish whether the alpha string is a multiword. If it is not an M-word it is categorized as an A-pattern. Although we are able to classify substrings of alpha patterns into various classes as explained in Table 3.3, we focus mainly on three general categories when training and guessing. Both classes of A-word and A-pattern require words from the dictionary when generating guesses. The attack dictionary usually contains both English words and words that may not be part of any language, but they are important in passwords. For

that reason we combine these two categories into one. The situation with R-word and R-pattern is very similar. When generating guesses for R-word or R-pattern, we take a word from the dictionary and we repeat it. Therefore, combining these two categories also make sense in our approach.

In order to understand how the new categories of alpha strings are incorporated into the context-free grammar, we give a simple example of deriving the grammar from the following small training set {lovelove123, security123, wordword456, iloveyou456, lovelove!!, wordword88}. Let R represent an R-word, A represent an A-word, and M represent an M-word. In order to add these components into the grammar, we first consider the derivation from S to the base structures as before (L, D, S, K) and then derive the subcategories from the L-structure. The above example would have grammar constructs as in Table 3.4. This grammar would derive the string $S \rightarrow L_8D_2 \rightarrow R_8D_2 \rightarrow loveloveD_2 \rightarrow lovelove88$, with probability $1/6 \times 4/6 \times 1/2 \times 1 = 2/36$. Note that here the derivation with an L-structure on the left hand side to its possible sub-patterns is done independent of the context. This approach allows us to consider as much as possible larger sets of passwords on which to determine the probabilities. By looking at all passwords that are L-structures, we have a fairly large set of passwords going to the subcategories.

Table 3.4 Example of Derivation for Alpha Strings

Left Side	Right Side	Respective Probabilities
S →	L_8D_3 | L_8D_2 | L_8S_2	4/6 1/6 1/6
L_8 →	R_8 | A_8 | M_8	4/6 1/6 1/6
D_3 →	123 | 456	1/2 1/2
D_2 →	88	1
S_2 →	!!	1
R_8 →	lovelove | wordword	1/2 1/2
A_8 →	security	1
M_8 →	iloveyou	1

In the guessing phase, when facing the A category, we simply replace it using words from the attack dictionary as before. In the R category we modify the use of the dictionary to double each word in the dictionary. For the M category the replacements come directly from the

grammar. Note that the grammar also has information about capitalization of the L-structures and we continue to use that information to create different masks for all of the subcategories.

3.2.1.1 The Multiword Algorithm. The M-word algorithm is rather more complex as generally there are many ways for segmenting a multiword into component words. We tried many different versions of breaking up the words such as finding the first leftmost substring that is a word in the dictionary and recursively calling this function for the remaining string. If this does not result in a multiword decomposition, we try the next leftmost substring that is longer than the first one we tried. We have also tried finding the longest initial substring that is a word in the training dictionary and identifies this as a possible first component by starting at the rightmost character of the string. If this does not result in a multiword decomposition, we try with the next shortest initial substring that is a word. When looking at the result of each of these approaches, there are multiwords that have not been broken down correctly since most of the times there are two or more valid ways of segmenting a string. For example, the string "bookstore" can be broken down into "book, store" or "books, tore". However, for an English spoken person, one is more preferable to another. We therefore developed an algorithm for finding the best breakdown of multiword by first identifying all possible breakdowns of each string. We then calculate the probability of each segmentation by multiplying the probability of each word in the segmentation found using a corpus of the most frequently used words in TV and movies scripts [37]. The scores in the list are representation of how often the word has been used or seen in TV and movie scripts. We therefore normalize these scores to use as probability values. If a word is found in the dictionary and there is no associated probability value in the scoring file, we assign the least probability value found in the corpus to the word. We then multiply the probabilities of each word in the segmentation. We identify the segmentation that has the highest score as the best possible segmentation in our algorithm. This is in fact a common technique in Natural Language processing [42].

In order to compare the algorithms and to understand which one works better we needed to develop ground truth to test our approach. We manually looked at 1000 alpha string components of our revealed password sets and identified the correct segmentation of each alpha string. We found that using the scoring algorithm we reduced the error rate of wrong segmentation of multiwords by 83% compared to our initial approach algorithm. However, we

had also seen errors related to classifying single words incorrectly as multiword. These problems are not only related to the segmentation algorithm, but also depend on the training dictionary used. We found that there are too many short words in our training dictionary that resulted in many strings misidentified as multiword. In the worst-case scenario, if all letters of alphabet exist in the training dictionary, every string would be considered as a multiword. We also found that there were many names that we were classifying as multiword simply because those names did not exist in our training dictionary. We improved our training dictionary by eliminating short words that do not make sense as words in English, as well as adding common proper names in different languages. Our results indicate that we were able to reduce the error rate of incorrectly classifying single words as multiwords by 71%. See our segmentation algorithm in Appendix A.

Veras et al. [28] presented a very interesting framework for semantically classifying passwords as described in section 2.1.4. Their work is the only work related to our approach, and yet there are many differences from ours. In terms of segmentation, while we also use a source corpus (which we refer to as a training dictionary) we only look for words and multiwords within parts of the password that consist completely of letters (alpha strings), whereas they look at the whole passwords including words and gaps. Our approach substantially simplifies both the resulting grammar and the segmentation algorithm and makes the guess generation algorithm simpler. Our scoring is based on unigrams and is in practice very fast. Their scoring model is more complex using back-off algorithm that starts with trigrams, and then bigrams, and unigrams. Although this might be useful and is a common approach in natural language processing problems, however it seems that bigrams and trigrams are infrequently used in passwords. Veras et al. [28] show the result of their tagging of the Rockyou password set that reveals the presence of trigrams and bigrams is 6.09% compared to 89.82% for unigrams.

Their approach as they discuss it clearly has a performance bottleneck and they speculate that it may be because they are generating many duplicate guesses since their grammar is ambiguous. We have focused on maintaining unambiguity of the grammar and we noticed no performance bottleneck when using our approach. In next section we present the result of our password cracking using multiwords in the grammar and we show that we have been able to generate more than 86 billion guesses without loss of efficiency. Another difference between our approaches is that they generate their guesses and word from the training set whereas we use an attack dictionary. The dictionary approach permits both quantization (many terminals can have

the same probability) as well as flexibility (the grammar does not have to be generated again when using a different dictionary). Finally, learning and using capitalization required adding mangling rules in their approach whereas we simply apply the capitalization to mutliwords as a mask when generating guesses.

3.2.2 Testing and Result

In this section we describe our experiments to show the effectiveness of learning keyboard and alpha string patterns. We tested all possible combinations of adding multiwords M, repeated words R as well as smoothed keyboard K to the grammar and show the cracking result of each combination in Figure 3.6. It is interesting to see that two of the combinations (adding repeated words R only, and adding repeated words along with keyboard patterns KR), perform worse than PPC. When comparing the addition of keyboard K to PPC, the performance is not very good in the very beginning (up to 35 million guesses), however the average improvement is significant and is 3.5%. The remaining four combinations (M, KM, MR, KMR) show a consistent improvement over PPC. Overall when learning multiwords (M, KM, KMR), the cracking result is always significantly better than PPC.

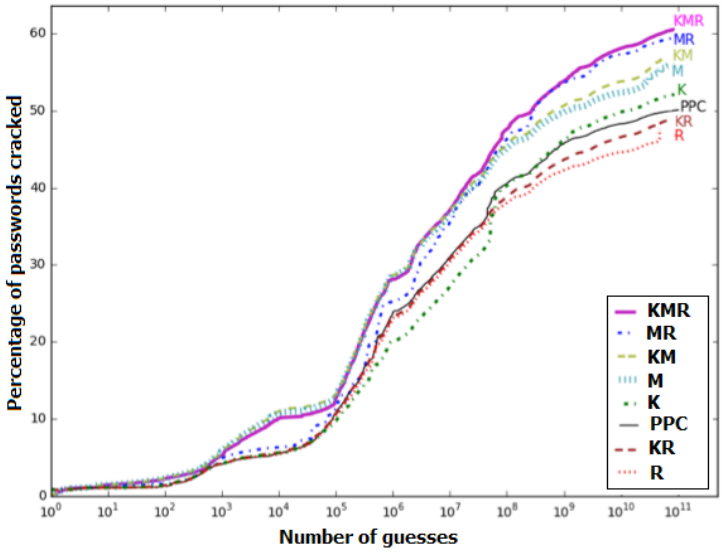


Fig. 3.6 Comparing Grammars with Keyboard and Multiwords using Combined-set in Log Scale

The results show that adding the Alpha Grammar results in a substantial improvement in the cracking over PPC. However, learning both classes of Keyboard and Alpha patterns is better than each alone. These results are consistent over the whole cracking curve. We also repeated these tests on Yahoo-set and CSDN-set. The results were very similar with Keyboard Alpha clearly the best over the cracking curve. We thus learn all three patterns (Keyboard, Multiwords and Repeated words) in our NPC system.

In Figure 3.7 we compare NPC against other password crackers: PPC and John the Ripper. The result show that NPC is substantially more effective as compared with both PPC and John the Ripper over the full cracking curve. The improvement of NPC over John the Ripper ranges from 13% to 305%. The improvement of NPC over PPC ranges from 15% to 22%.

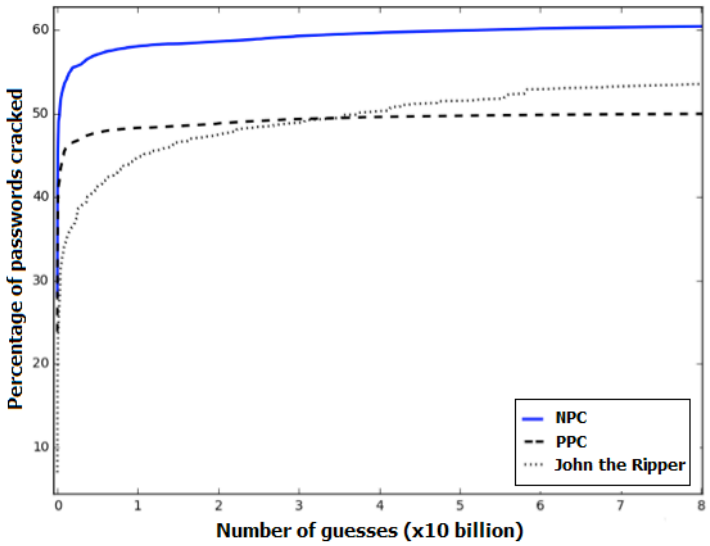


Fig. 3.7 Comparing Password Crackers using Combined-set

This is most clearly seen in Figure 3.8 where we plot the improvement values as shown in the legend. The dashed line shows the consistency of our improvement over the original PPC. We also tested NPC against the other two using both Yahoo-set and CSDN-set. The results were similar to those for Combined-set. For Yahoo-set the improvement of NPC over PPC ranges from 1% (in the very early stage) to 16%. For CSDN-set the improvement of NPC over PPC ranged from 14% to 17%. When comparing against John the Ripper, we first consider a late stage of 20 million to 85 billion because John the Ripper does quite poorly in the early stage. In

the late stage, the improvement of NPC over John the Ripper for Yahoo-set ranges from 35% to 152% and for CSDN-set the improvement is 84% to 264%.

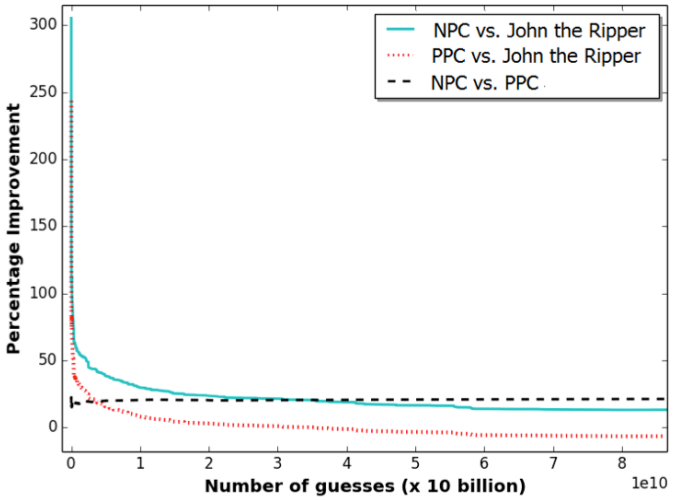


Fig. 3.8 Improvement of Crackers Against Each Other using Combined-set

We also compared against Hashcat using two of its best rule sets: Best64 and Deadone (d3ad0ne) [22] in Figure 3.9. Since these two rule sets are quite small, the guesses generated were only 64.5 million and 21 billion respectively.

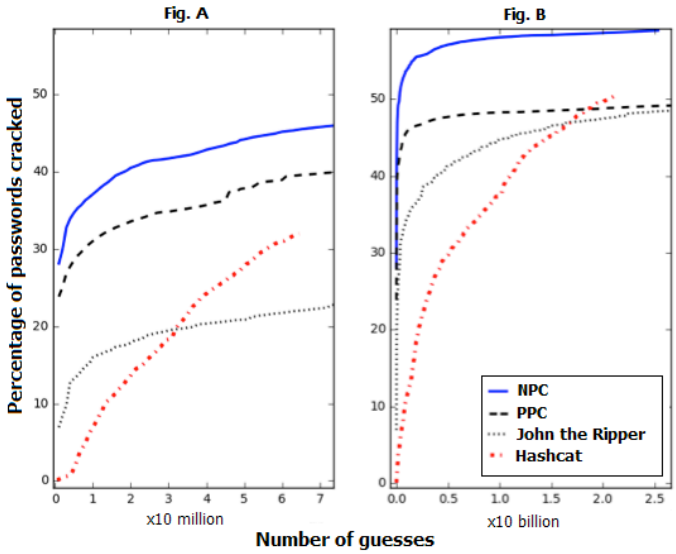


Fig. 3.9 Comparing Password Crackers using Combined-set: A) Hashcat using Best64 Rule Set. B) Hashcat using Deadone Rule Set

Figure 3.9A shows the comparison of the Best64 rule set with the other three password crackers and Figure 3.9B compares the Deadone rule set. Clearly NPC is extremely dominant in both figures. The improvement of NPC over Hashcat at the end of the Hashcat cracking with Best64 is 42% and with Deadone is 16%.

3.3 Attack Dictionaries

3.3.1 Background and Motivation

In dictionary-based attacks different mangling rules are applied on a list of words called an attack dictionary to create password guesses. Therefore in order to correctly guess a password not only we need to apply the right mangling rule but we also need to include the right word in the dictionary. The probabilistic password cracker derives the mangling rules from a training set of real user passwords and has been shown to be very effective. We also showed in previous section how we were able to improve the grammar by learning more patterns. However, in order to be successful, we still need to explore how to best choose the attack dictionaries. The size and content of attack dictionaries can affect the probabilities and the guesses and the order they are generated, which in turn can affect the efficiency of our cracking.

The dictionaries used in password cracking are usually a list of common passwords that have been cracked previously or a list of English words that have been experimentally shown to be effective. Although some common password lists exist (derived from passwords cracked or disclosed passwords), there are few studies showing the effectiveness of such lists for probabilistic context-free grammar based crackers. Dictionaries are sometimes viewed as the guesses themselves. For example, Bonneau [43] creates dictionaries for different groups of Yahoo users based on linguistic background and defines a dictionary as the top one thousand actual passwords from that group. The author then determines the effectiveness of such dictionaries against other linguistic groups.

Using a dictionary of actual guesses is quite different from finding a good dictionary to be the base for the alpha string substitutions as used in PPC. For example, in PPC doubling the size of an attack dictionary is not a cost problem in terms of the size of the dictionary with respect to cracking as such. Guesses will still be generated quickly and in highest probability order. However, the probability of the guesses will change and thus the guesses will be tried in a different order. Furthermore, new combinations would likely be tried (good, because there are

more alpha words to replace) but too many words for the same base structure might reduce the probability of each of the terminal guesses in a base structure so they might not be tried until much later (possibly bad).

In other studies, dictionaries are used both as a source of passwords as well as a source for generating variant guesses by applying mangling rules [44]. Dell'Amico et al. [45] evaluate several dictionaries available from John the Ripper by first comparing the passwords cracked using the dictionary entries only. Their results show that it is better to use the same type of dictionary as the target type (for example Finnish when attacking Finnish passwords) and although larger dictionaries are better, there are diminishing returns when using these larger dictionaries.

Our goal, in this work is to investigate how different dictionaries can be effective as the probabilistic password cracking system generates more and more guesses. Note that dictionaries in PPC are used only to replace alpha strings in the grammar. In this way a dictionary can typically generate more passwords that could feasibly be tried even in extremely long cracking sessions. Thus, the full set of guesses that a dictionary can produce is only partly relevant to its effectiveness. There are also many things that can affect the efficiency of a dictionary in the probabilistic context-free password cracking approach. For any base structures in NPC containing A-patterns or R-patterns, all words of the same length from the dictionary are going to be tried at the same time because of the assumption that all words of the same length from one dictionary have equal probability. With a larger dictionary, trying more words at that point in time delays trying other combinations. However, with a larger dictionary, because the probability value of each word is smaller the base structure itself might be tried much later. On the other hand, a very small dictionary might not be effective at all because it will obviously reduce the variety and number of guesses.

NPC also has the capability of using multiple attack dictionaries when cracking passwords. Probability values can be assigned to each dictionary and therefore different probability values to sets of words. The end probability values of the actual words do not only come from the probability value assigned to the dictionary containing the word, but multiplies with $1/n_L$, where n_L is the number of words in the dictionary of length L. Having this capability one can have a fairly large dictionary with lower probability words and a list of common passwords with higher probabilities assigned to them as a secondary dictionary. This way we can

first try the most probable words with different mangling rules and try the other possible words that are not so common at a later point. In NPC, the number of dictionaries corresponds to defining the number of equivalence sets of probabilities of words of each specific length. Note that when multiple dictionaries are used, even though they may have duplicate words, the final set of words and their probabilities used in cracking have no duplicate words.

There have been no studies as far as we are aware that explore how to use multiple dictionaries effectively in probabilistic context-free grammar password cracking. The difficulty in regards to designing such studies is the number of variables that change at the same time in regard to dictionaries and more specifically with regards to multiple dictionaries in NPC. In fact when cracking, the number of dictionaries used, the weights assigned to each dictionary, the usefulness of actual words in the dictionary, as well as the probability values assigned to each word in each dictionary (which depends on the length of the dictionary and also depends on the number of duplicate words that exist in multiple dictionaries) can all affect the results. In the experiments section we discuss how we explored this problem space by trying to keep as many features as possible constant and varying only a few.

In the next section I explore how to improve attack dictionaries for NPC. First new metrics for comparing dictionaries are developed and then the results of the effectiveness of primary dictionaries as well as secondary dictionaries are presented in section 3.3.3. The improvements are very significant and could likely also be used to make the attack dictionaries more effective for other password cracking systems.

3.3.2 Measuring the Effectiveness of a Dictionary

The most basic question is how one can measure the effectiveness of one dictionary as compared to another. We developed an approach to measure the effectiveness of a dictionary by considering its coverage and precision with respect to a reference set (set of passwords).

Let W be a set of words $\{w_1 \ldots w_n\}$ that is going to be used as a dictionary and let R be a reference set of passwords $\{p_1 \ldots p_m\}$. A word w is *found* in R if it is an L-structure in at least one of the passwords. Let $I(w, R) = 1$ if w is found in R and $I(w, R) = 0$ otherwise. Precision of a dictionary W with respect to a reference set R is then defined as:

$$P(W,R) = \frac{1}{|W|}\sum_{i=1}^{n} I(w_i, R) \tag{3.2}$$

Assume a password p has k different L-structures in it. Let the *count c(w, p)* be the number of L-structures in p that have the value w. Coverage of a word w with respect to a password p (and naturally extended to R) is defined as:

$$C(w,p) = \frac{c(w,p)}{k} \qquad \& \qquad C(w,R) = \sum_{i=1}^{m} C(w,p_i) \tag{3.3}$$

We define R_L as the subset of passwords in R that have at least one L-structure. Coverage of a dictionary W and reference set R is:

$$C(W,R) = \frac{1}{|R_L|} \sum_{i=1}^{n} C(w_i,R) \tag{3.4}$$

We only consider the passwords that include L-structures because the dictionary has no relevance to cracking the passwords that have no L-structures. Note that C(W, R) and P(W, R) are values between 0 and 1. Precision is a measure of how compact the dictionary is for the reference set. For an ideal precision measure of 1, a dictionary should only consist of all the words that appear in the reference set. Coverage measures how useful the words of a dictionary might be for potentially cracking passwords in a target set. For an ideal coverage measure of 1, every L-structure of the reference set should be a word in the dictionary. We define a perfect dictionary (D_R) for a reference R as the set of all words that appear in R. This perfect dictionary has both coverage and precision equal to 1 and the words in the perfect dictionary can be ordered by their individual coverage values *C(w, R)*.

3.3.3 Testing and Result

As discussed previously, NPC can use multiple dictionaries. Typically, a primary attack dictionary and a smaller secondary dictionary are used. In our tests we first explore the effectiveness of different primary dictionaries based on the metrics defined in the previous section. We then consider using various secondary dictionaries to give higher probabilities to a selected set of words and explore the additional utility on the success of the cracking.

3.3.3.1 Primary Dictionaries. In this section we compare different attack dictionaries and show how to create more effective ones using our metrics. Since we have been using dic0294 in our testing we use this as a base for our comparisons and improvements. This dictionary has strings containing digits and special characters, which we had removed. This results in a dictionary of size 728,216. We created a dictionary from the English language set (containing only alpha strings) of about the same size from John the Ripper's wordlist collection

[21] (Jtr_En). We also created a dictionary of a similar size from 2.5 million randomly chosen Rockyou passwords by stripping out the alpha string components and removing duplicates. The coverage and precision of each of these dictionaries with respect to Combined-test (reference R) are shown in Table 3.5.

Table 3.5 Coverage and Precision with Respect to Combined-test

Dictionary	Size	Coverage	Precision
Rockyou dict	728,376	0.74	0.11
dic0294	728,216	0.55	0.06
Jtr_En dict	728,749	0.49	0.05

We ran a password cracking session with each of the dictionaries against Yahoo-test in Figure 3.10A and Rockyou-test in Figure 3.10B. The results show that the cracking curves are consistent with the precision and coverage metrics, with better rates of cracking for dictionaries having higher coverage/precision. Note that the Rockyou dictionary has higher coverage since it is calculated with respect to Combined-test, which contains mostly Rockyou passwords. Thus, this dictionary may not be a good candidate to use as a generic dictionary for other target sets.

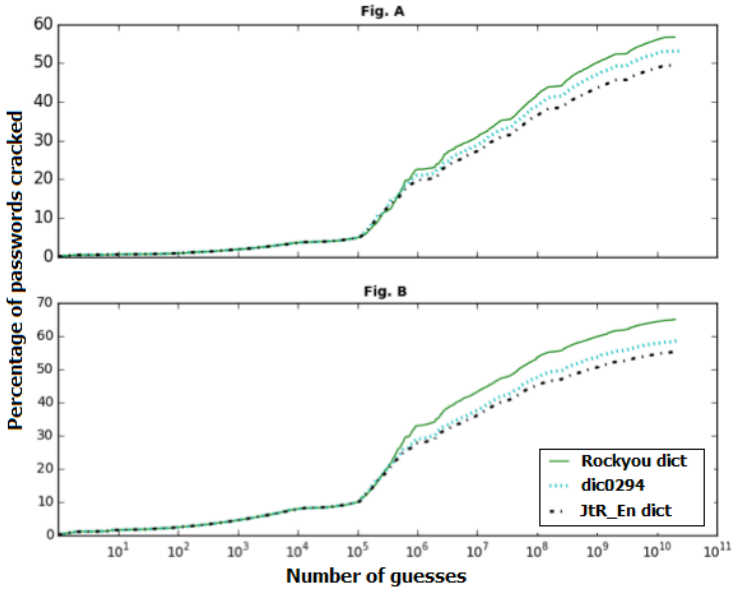


Fig. 3.10 Primary Attack Dictionaries with Different Coverage and Precision in Log Scale A) Using Yahoo-test as Target B) Using Rockyou-test as Target

We next created different dictionaries from dic0294 by systematically altering coverage and precision to see how the cracking result changes. In our first series of experiments we used the baseline dic0294 and calculated its metrics with respect to the reference Combined-test.

$$C(dic0294, R_{Combined}) = 0.55, \qquad P(dic0294, R_{Combined}) = 0.06$$

We then created two dictionaries as variants of dic0294, increasing the coverage to 0.7 and 0.9 respectively without changing the precision. We call these variants dic0294_c70 and dic0294_c90. The sizes of these variants increased to about 1.56 million and 2.58 million respectively. To increase the coverage of a dictionary D with respect to a reference R, we added words from the perfect dictionary D_R. Note that optimally achieving a specific coverage value is actually a Knapsack problem [46] but the heuristic of adding words in highest coverage order works fairly well in this case. Let n_r be the number of words added from D_R. To maintain the precision P we also need to add n_n words that are not in D_R, where:

$$n_n = n_r(\frac{1}{P} - 1) \tag{3.5}$$

Since in cracking we would not know the actual target set, we explored the use of the metrics derived from reference Combined-test by testing how well the derived dictionaries would do on the targets Yahoo-test shown in Fig. 3.11A and Rockyou-test shown in Fig. 3.11B. In this experiment we trained on Combined-training using our new system NPC.

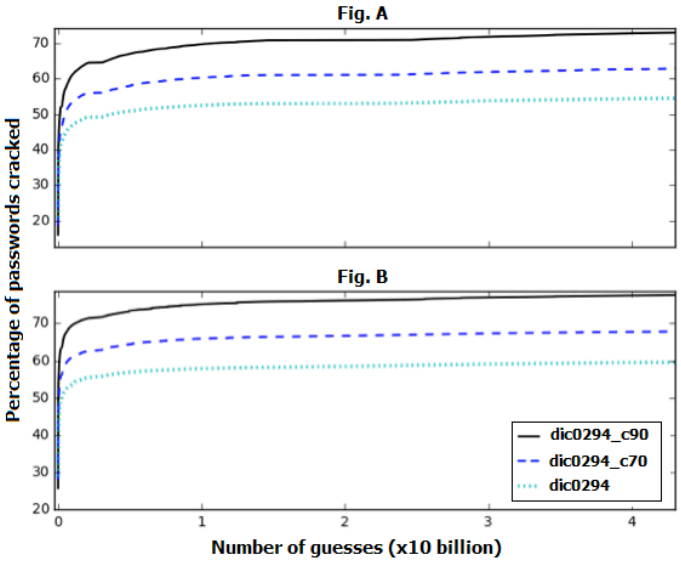


Fig. 3.11 Dic0294 Variants with Precision Fixed at 0.06: A) Using Yahoo-test as Target B) Using Rockyou-test as Target

The results were remarkably good and support the premise of our metrics. In Figure 3.11A, the average improvement when using dic0294_c90 over dic0294 over the entire cracking curve is 33%. Similarly, in Figure 3.11B the average improvement over the cracking curve is 30%. Not only that, we subsequently checked the coverage metrics relative to the new targets and found that coverage on one test set seems to map appropriately to coverage on the different target sets. For example, although the initial coverage for dic0294_c90 was derived from Combined-set (90%), its coverage when measured on both targets is very similar. See Table 3.6.

Table 3.6 Coverage and Precision for Target Sets

	Yahoo-test		Rockyou-test	
	Coverage	Precision	Coverage	Precision
dic0294	0.57	0.037	0.54	0.03
dic0294_c70	0.71	0.028	0.69	0.02
dic0294_c90	0.9	0.025	0.89	0.02
dic0294_p10	0.53	0.051	0.52	0.04
dic0294_p20	0.50	0.087	0.5	0.075

We also did an analogous series of tests on the same targets where we kept the coverage of dic0294 at the baseline and created two other variant dictionaries dic0294_p10 and dic0294_p20, increasing the precision to 0.1 and 0.2 respectively. The results are shown in Figure 3.12. In order to do this, we removed words not in D_R from the dictionaries and their sizes decreased to about 450K and 225K respectively. We expected that the higher precision dictionaries might do better in cracking but they actually did worse, because their coverage with respect to the targets decreased. See Table 3.6. This indicates to us that coverage is extremely important and is more important than precision.

The results of these tests with attack dictionaries show that our metrics for measuring dictionaries can be extremely useful in creating and comparing dictionaries. The algorithms to improve the coverage and precision of the dictionaries have also been implemented as part of our NPC system. Note that these results also shed light on the questions regarding the size of the dictionaries. It is often stated that there are diminishing returns from larger dictionaries. The results of our tests seem to indicate that if a larger dictionary is created in the manner we recommend the cracking improvement is certainly substantial.

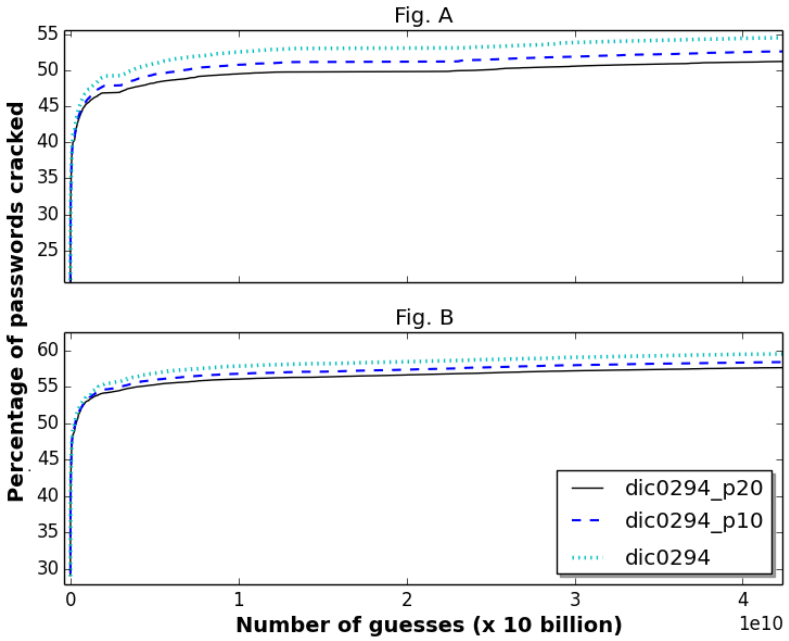


Fig. 3.12 Dic0294 Variants with Coverage Fixed at 0.55: A) Using Yahoo-test as Target B) Using Rockyou-test as Target

3.3.3.2 Secondary Dictionaries. In this series of tests we explore secondary dictionaries by using NPC trained on Yahoo-training and cracking Yahoo-test. We use dic0294 as our primary dictionary. We use three secondary dictionaries: (1) *common-passwords* (which is presumably an "optimized" dictionary) that contains 815 words; (2) *TopWords* from Yahoo-training or same set (a list of the highest frequency A-patterns found in our training set) also of size 815; and (3) *TopWords* from Combined-set. We assign probability 0.9 to the primary dictionary and 0.1 to the secondary dictionary. Note that the probability values assigned to the dictionaries actually give higher weight to the words in the secondary dictionary. Since the primary dictionary has far more passwords than the secondary dictionary, $1/n_L \times 0.9$ in the primary is still a fairly small number compared to the probabilities of the words in the secondary dictionary.

Figure 3.13 shows the results of our tests where we tested dic0294 with and without secondary dictionaries. The results show that using a secondary dictionary of top words from the same set is more effective than all the others, and that even if such a word list is not available, creating a secondary dictionary from another revealed password set can improve the cracking almost as well. The question then was that whether this was due to differential weights for

certain words or whether the secondary dictionary was adding new words that are not in the primary dictionary. Further analysis on the secondary dictionaries showed that all words in the common-passwords list and also all words in the TopWords list happen to be already included in dic0294. We concluded that the improvement when using a secondary dictionary is not because new words have been added, but because sets of words are given higher probabilities.

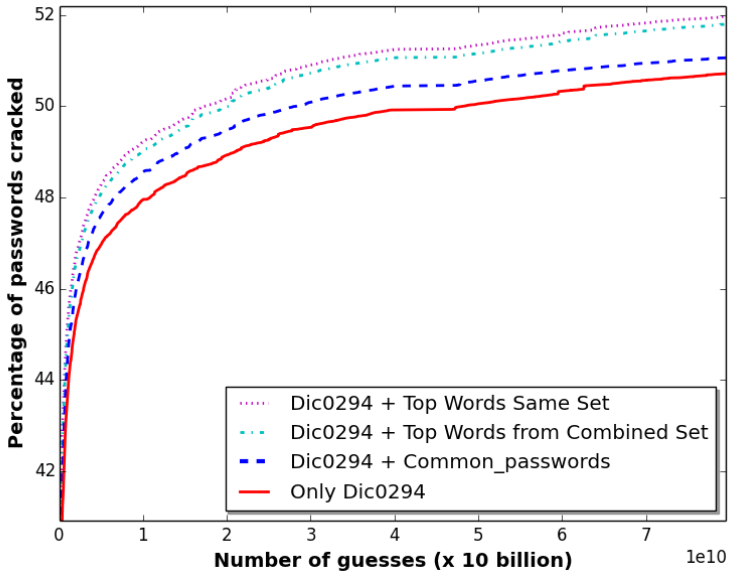


Fig. 3.13 Cracking Yahoo-set with Several Secondary Dictionaries

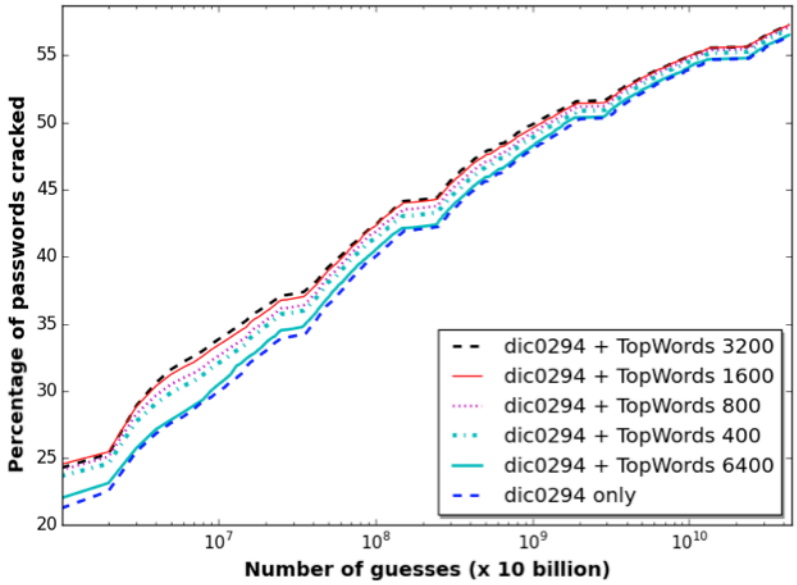


Fig. 3.14 Varying the Sizes of the Secondary Dictionaries Cracking Yahoo-test in Log Scale

We next tested different sizes of the secondary dictionary. We created different sizes of TopWords by selecting 400, 800, 1600, 3200 and 6400 of the highest frequency words from our training set. For our primary dictionary we added the largest TopWords to dic0294 so that all words from the secondary dictionaries are included in the primary dictionary as well. This way we ensure that the reason for cracking improvement is the way the probability values are assigned to each word. One might think that the larger the secondary dictionary, the better the results might be, particularly since the secondary dictionaries are all fairly small. This was true for sizes up to 3200. However as seen in Figure 3.14, at size 6400, the advantage of giving higher probabilities to some sets of words no longer exists and at this level it becomes virtually equivalent to not using a secondary dictionary at all.

3.4 Final Testing of NPC Compared to Other Password Crackers

In this section we compare the proposed NPC system against PPC and two other recent password crackers. Our NPC system combines all of the advancements we have proposed: new patterns, and improved primary and secondary attack dictionaries. Figure 3.15 shows the result of comparing NPC using the dictionaries dic0294_c90 and TopWords3200 versus PPC using dic0294 and common-passwords in log scale. We use Combined-training and test against Yahoo-test in order to also show the effectiveness of both crackers when not having the advantage of training on a set that is similar to the target set.

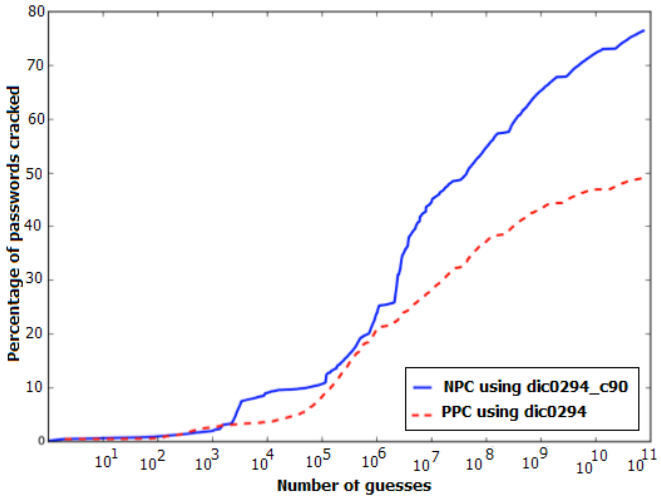


Fig. 3.15 Results of NPC with Combined-training and Yahoo-test in Log Scale

At the end of the cracking run, NPC has cracked 76% of the passwords and shows an average improvement over PPC of 55%. With respect to the effectiveness of NPC in the early part of the cracking curve, we cracked 70% of the passwords within 5 billion guesses, which translates to under three hours on a regular laptop. Table 3.7 shows the number of base structures and other components for Combined-training set in both NPC and PPC. As can be seen NPC creates many more base structures and yet there is no performance bottleneck.

Table 3.7 Numbers of Components in Grammars Created by NPC and PPC

Approach	# Base Structures	# Digits	# Special Symbols	# Keyboards	# Multiwords
PPC	7,650	82,237	608	0	0
NPC	201,019	82,237	608	47,455	133,364

We also compare NPC with the semantic approach of Veras et al. [28] and the Markov approach of Ma et al. [25]. Unfortunately, standard benchmark suites are not available for comparing algorithms against well-defined training and test sets when researchers use different algorithms. We do not believe it is fair to compare results of our work to algorithms of other researchers by creating versions (probably inferior) of their algorithms ourselves. For the comparative tests against these approaches we chose to approximate their training and test sets as closely as possible and report our results against their reported results.

Veras et al. [28] used a semantic approach and a context-free grammar that lets them explore passwords containing multiple words or phrases. Since we did not believe that we could faithfully recreate their algorithm, we chose to run our NPC on a similar training and test set as they did and superimpose our cracking curve on their reported graph (their figure 3 in [28]). We trained on 2 million Rockyou passwords and tested on Myspace [31]. We improved our JtR_En dictionary with respect to a set of one million passwords from Rockyou to reach the 90% coverage and used this dictionary as our primary attack dictionary. Figure 3.16 shows the comparison on a cracking run of only 3 billion guesses (the maximum number they report on). As can be seen our cracking results are much more comparable to their best efforts than was PPC.

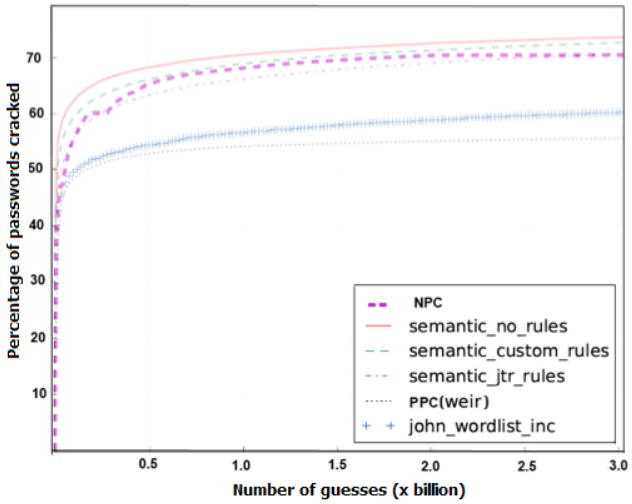


Fig. 3.16 Comparing NPC with the Reported Results of Figure 3 of Veras et al.

The paper by Ma et al. [25] indicates the importance of using guess numbers for comparing different models. So in order to do a comparison with the Markov approach we again decided to run NPC on a similar training and test set as they used and superimpose our cracking curve on their reported work (their figure 2b in [25]). We used Rockyou training and the same test set that Ma et al. reported using (Yahoo + PhpBB). Figure 3.17 shows that our approach is comparable to the best Markov approach that they considered. Note that their guess generation is limited to about 15 billion guesses.

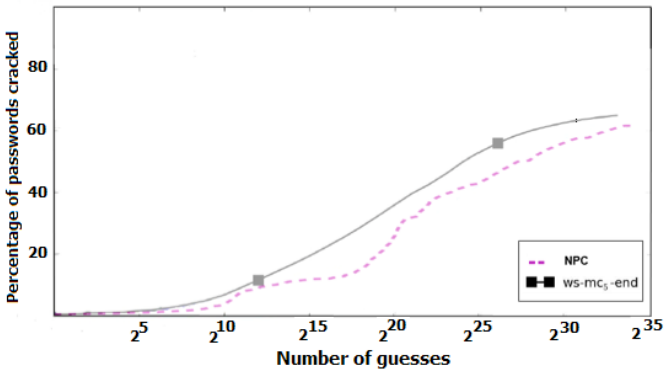


Fig. 3.17 Comparing NPC with the Best Markov Model Reported in Figure 2b of Ma et al.

CHAPTER 4

TARGETED PASSWORD CRACKING

4.1 Background

With respect to password security, it is not only essential to have a secure system to store user's passwords, but it is also important how users create and use their passwords. The number of accounts for a single user is growing. The result of a survey of 2000 users has shown that a typical user has about 25 online accounts and one in four user uses a single password for most of their accounts [47]. Florencio et al. [48] showed that on average a user has 6.5 passwords and each password is typically being reused across 3.9 different websites. Enforcing complex password policies makes it harder for users to create memorable passwords. Because of this, many users reuse the same password for multiple accounts against experts' advice. This reduces the security tremendously since when an attacker obtains a password, it is often tried on many different websites. Thus no matter how secure a service is; the security of it can be reduced because of its users' actions. As more and more websites replace usernames with email addresses, it becomes much easier for attackers to attack and access our accounts. Users are often forced to change their password on a given account because of a threat or simply due to expiration policies. In these situations users are more likely to apply only slight changes to their previous password instead of creating a new one. Furthermore, users also tend to use a password with slight modification across different websites. Having different password creation policies for different websites might prevent users from some reuse of the same password (an unintended consequence), but it does not prevent users from using passwords that are very similar. A study by Shay et al [49] conducted on 470 University student, staff and faculty has shown that 60% used one password with slight changes for different accounts. In [50] the authors examined leaked password sets and found that users often do simple tricks to slightly change their passwords and to work around different password policies.

In this chapter, I explore how to use the information about targets to help crack their passwords. Information could be names of family members, important dates or numbers, as well as any of their previous passwords. By modeling the differences between two or more old passwords, I show how to find their new password under the assumption that users often modify

their passwords by slight changes. A necessary assumption is that a set of one or more password sequences with slight changes is available. Clearly having more data on changed password sequences helps in better modeling the differences between passwords and predicting the new one. However, the first problem we encountered was the lack of data in this domain. There is not enough data available that contains changed passwords of users. Das et al. [50] used publicly available leaked password sets with user identifiers and analyzed the data to find passwords for the same user. They were able to find 6077 unique users with at most two passwords for each, from those about 43% were identical passwords and the rest were non-identical. Although this data could be representative of similar passwords for a specific user, it is not useful when analyzing the specific changes users make to their passwords for one account. Perhaps the closest study to the work in this chapter is Zhang et al. [51], which is a large-scale study based on password changes necessitated by password expiration. The authors were able to obtain a dataset of over 7700 accounts for which they had a known password and a subsequently changed password. They modeled a password change as a sequence of transforms (based on several different criteria) and organized these transforms as a tree from the old password as root. A path in the tree is a sequence of transforms that yields the new password with common subsequences being the same from the root. A search starts from the root with an input password and upon visiting each node in the tree the corresponding transform is applied to the output of the parent node. Then each output is tested as a password guess against the target password hash. One of the main difficulties of this algorithm was the high time complexity of the search algorithm to effectively walk the tree from its root. In their work the depth of the tree was limited to at most 3.

4.2 Collecting Data and Survey Result

In order to collect data for testing I developed a survey that required users to create and change passwords. The website was created using python and html5 to host the surveys. The actual programming for the website was mainly done by Ryan Kuhl and later edited by Frank Valcarcel. First time users click on a link, are presented with a consent form, and upon consent are asked to create an account using their FSU email address. They need to create a password for the account and are informed that their passwords will be saved and analyzed. The only policy on the password is to have at least 8 characters. After they create their accounts they are asked a few survey questions. The second time the participant visits our website, the user must login to

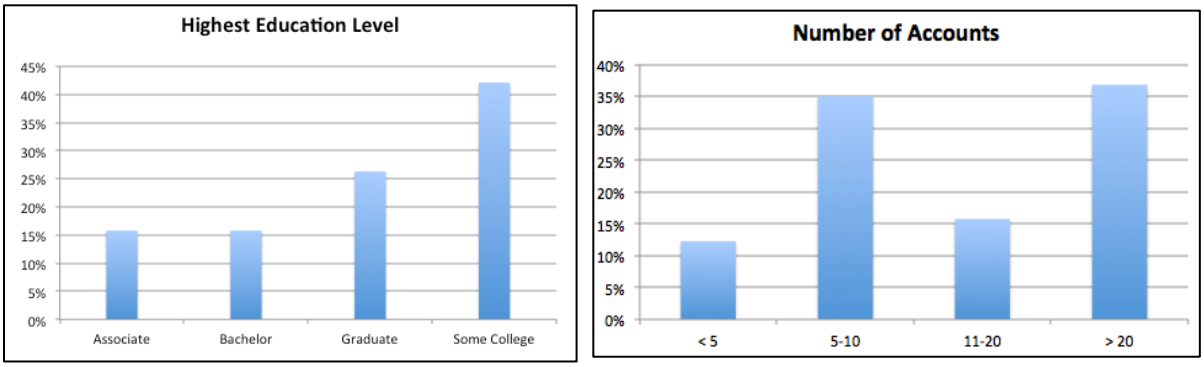
the previously created account with the password previously created and answer another series of questions. For the third site visit, the participant is asked to change his/her password and then answer another series of questions. Finally, the fourth visit completes the survey through the user logging in with the changed password and completing a set of questions. We enforce that users must not login again each time until the next calendar day and the total time to complete the survey was limited to about a week to ten days. The reason behind multiple logins is for the users to get more familiar and comfortable with the password they first created before asking them to change it. If the users forget their selected passwords, they can use the *forgot password* link on the page and their password will be sent to them via email. See the survey questionnaire in Appendix B. See also the Human Subject Approval letters in Appendix C and a sample of consent form in Appendix D.

4.2.1 Survey Result

We used Florida State University (FSU) students in our survey study since we could easily control who is participating and whether each individual is participating only once by enforcing the use of their FSU email addresses when creating their accounts. We sent an email to all students in the department of Computer Science and to a list of about 2000 randomly selected students at Florida State University asking them to participate in our survey study. For this study 144 students created accounts, 56 of whom changed their passwords but did not necessarily complete the last survey. 50 students completed all four steps required. Recall that each time users log in they are asked a few questions. In this section we next discuss and analyze their responses. We mainly present the result of the 56 participants who changed their passwords, except for the last survey question in which only 50 answered the questions.

In this study, 53% of our participants were female and 47% male. 68% of our participants were between the ages of 18 and 24. 17% were in the range of 25-34 and 13% were 35-44 years old. 50.88% of participants were majoring in computer science or related field. 67% of our participants have been using computers for more than 10 years and 31% of them were using it for about 6-10 years. Figure 4.1 (a) shows the highest education level of the participants, in which about 25% of participants were graduate students and the rest were undergraduate students. Figure 4.1 (b) shows the number of accounts they have. About 35% of participants

indicated that they have 5 to 10 accounts. More than 35% also indicated that they have more than 20 different accounts.



(a) (b)

Fig. 4.1 Result of Survey Questions: (a) Highest Education Level (b) Number of Accounts

We asked the participants whether they create unique passwords for each account and the results are shown in Figure 4.2. About 40% responded that they do not create new passwords and that they use their old passwords. Only 14% claimed that they create new passwords for each account and the rest most of the times create new passwords but sometimes use their old ones.

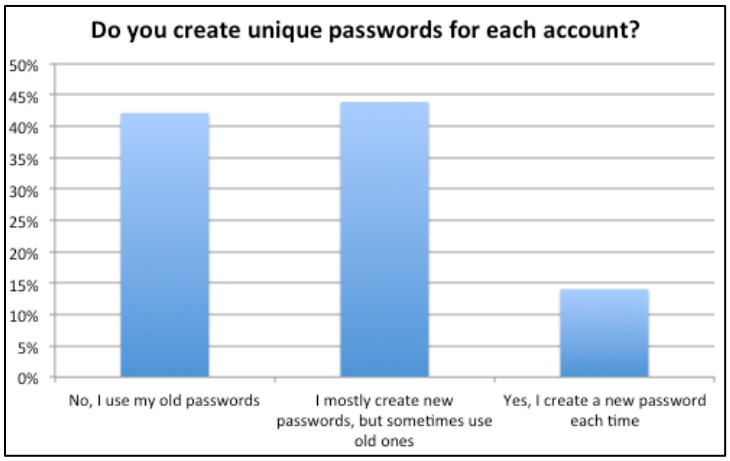
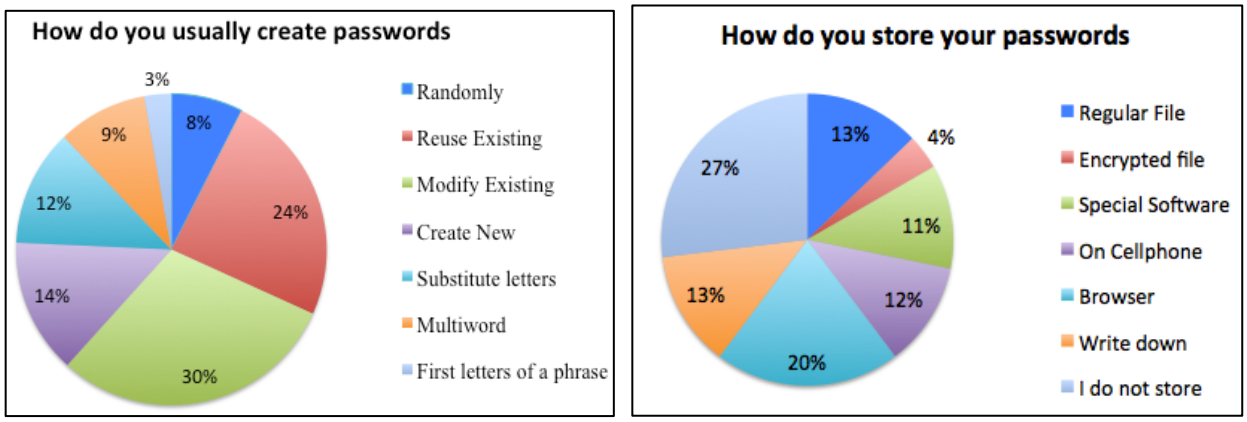


Fig. 4.2 Result of Survey Question: Do you Create Unique Passwords for Each Account

In Figure 4.3 (a) we asked our participants how they usually create their passwords. The result show that 30% modify their existing passwords to create a password, about 24% reuse

their old passwords and only 14% create new passwords. This is consistent with other reported studies and shows that our approach can be very useful in attacking a lot of passwords since many users do reuse and modify their old passwords. In Figure 4.3 (b), we see the result of how users usually store their passwords. About 13% store their passwords on regular files on their computers without any encryption, and 12% store it on their cellphones. 73% of our participants store their passwords in some way. This statistic is also helpful for us in another way as in chapter 6 we develop a tool that can identify passwords stored on hard disks and cell phones. That shows how vulnerable users can be by following current habits.



(a) (b)

Fig. 4.3 Result of Survey Questions: (a) How Do you Create Passwords (b) How Do you Store Passwords

4.3 Modeling the Differences

In this section we discuss our approach for integrating the information about the target into the probabilistic context-free grammar. We later show that using this context-free grammar we are able to predict and crack new passwords of the same user. Our approach consists of two different methods. The first approach is used when only one old password of the user is accessible. We discuss this approach in section 4.3.1. In the second approach, our system has access to at least two different passwords as a sequence and learns the changes made between these two passwords and uses the information to predict the new password. We discuss the latter approach in section 4.3.2.

4.3.1 Using AMP Distance Function to Create a Grammar

In this approach, the only available information is the user's previous password. Based on that, we would like to predict the new password or generate guesses similar to that. Following the assumption that users will most likely change their passwords with slight modifications, we use the AMP distance function [5] to generate guesses similar to the initial password.

In this approach we are not only interested in generating guesses similar to the initial password, but we would like to create a probabilistic context-free grammar for predicting the new password. AMP uses a distance function to create strengthened passwords within edit distance one of the user-chosen password and it was designed based on Damerau-Levenshtein edit distance. The AMP distance function includes insertion, deletion and transposition of components in the base structure of a password as well as insertion, deletion and substitution inside a component. The improved distance function with the addition of keyboard patterns and multiword is described in section 4.3.1.1. It starts with the old password as the root of a tree, and generates all possible passwords within edit distance one of the root. We then create a probabilistic context-free grammar for the set of similar passwords (within edit distance one of the initial password). This context-free grammar represents all the possible new passwords that can be created subsequent to the use of the old password and is called *EGrammar* for Edit Distance Grammar. In this work, we consider every possible change to be equally likely, but in the future, by training on large numbers of old and new password pairs, we may be able to give different probability values to different changes. We can view the probability values in the EGrammar as conditional probabilities p(y|x) when y is the new password and x is the old password. Thus, the probability values in the EGrammar could be the probability values conditioned on the input password.

Suppose the given password is "alice123!" with base structure $L_5D_3S_1$. Using operations defined in the AMP distance function, we create passwords within one edit distance of "alice123!". For example, we can insert an S_1 component between "alice" and "123" or we can delete "123". Similarly, we can insert digits in between 123 and create 1293 for example. Table 4.1 shows the full EGrammar for the given password "alice123!". As discussed before D stands for digits, S for special symbols, and C for capitalization (L: lowercase, U: uppercase). As shown in this example the grammar is very small compared to typical context-free grammars for password cracking; however it captures all edit distance variations of the given password.

Table 4.1 Example of EGrammar for the Given Password “alice123!”

Base structure	D_1	D_2	D_3		D_4		S_1		S_2			C_1	C_5
$L_5D_3S_1$	0	12	120	153	0123	1233	@	|	)!	!_	!|	L	LLLLL
L_5D_3	1	13	121	163	1123	1243	!	\	^!	!]	<!	U	LLLLU
$L_5S_1D_3$	2	23	122	173	2123	1253	?	.	!-	!=	@!		LLLUL
$S_1L_5D_3S_1$	3		123	183	3123	1263	/	_	!:	!^	!#		LLULL
$L_5D_3S_2$	4		124	193	4123	1273	}	#	=!	!!	!"		LULLL
$L_5D_4S_1$	5		125	023	5123	1283	:	$	!{	!(	!,		ULLLL
$L_5S_1D_3S_1$	6		126	223	6123	1293	+	]	_!	:!	#!		
$L_5D_3S_1L_1$	7		127	323	7123	1230	{	~	.!	!$	!%		
$L_5D_3L_1S_1$	8		128	423	8123	1231	*	>	{!	[!	!/		
$L_5D_3S_1D_1$	9		129	523	9123	1232	<	,	!'	!}	!)		
L_5S_1			103	623	1023	1234	(	=	'!	![	$!		
$D_3L_5S_1$			113	723	1223	1235	%	^	(!	!+	`!		
$D_1L_5D_3S_1$			133	823	1323	1236	"	'	!`	+!	!.		
$L_5D_2S_1$			143	923	1423	1237	)	;	?!	!~	&!		
					1523	1238	`	[	%!	!<	|!		
					1623	1239	-	&	*!	~!	!\		
					1723				!?	!*	]!		
					1823				!;	-!	}!		
					1923				!&	\!	/!		
					1203				,!	"!	;!		
					1213				!>	!@	>!		

4.3.1.1 Edit Distance Function. In this section we overview the distance function defined in [5] and we introduce the additional operations that we have developed in order to adapt the distance function to the new context-free grammar introduced in chapter 3 with the addition of keyboard and multiword patterns.

Operations on the Base Structure:

- **Insertion:** Inserting a component of length one is allowed only when it is not of a same type as its adjacent components. For example, if the base structure is $L_5D_3S_1$ we can insert D_1 in the beginning to make $D_1L_5D_3S_1$. There is no insertion of K_1 or M_1 since a keyboard or multiword component of length one is not defined. It is also possible to insert a D_1 or S_1 in between two words in a multiword. For example, for a password containing starwars (M_8), we can create star5wars (inserting a digit) or “star!wars” (inserting a special character).

- **Deletion:** deleting a component can be done if the number of components is not 1 and if it does not make two components of the same type adjacent.
- **Transposition:** Exchanging two adjacent components can be done only if it does not make two components of the same type adjacent to each other. For multiword components, we can also transpose two adjacent words, as well as the first and last word. For example, "mysweetbaby" can be changed to "sweetmybaby", "mybabysweet" and "babysweetmy".

Operations on the Component:

- **Insertion:** inserting one character of the same type inside a component is allowed. *Example*: if component D_3 = 123, we can transform it to **4123** by inserting 4 at the beginning.
- **Deletion**: deleting one character inside a component is allowed only if the length of the component is not equal to 1. For multiword components, we also allow deleting a word from a multiword which results in a new base structure as well as a new multiword in the grammar. For example, given password mysweetbaby12 with base structure $M_{11}D_2$, we can create other base structures such as M_9D_2, M_6D_2, M_7D_2 as well as "mysweet", "mybaby", and "sweetbaby" as multiwords in the grammar.
- **Substitution**: we can substitute a character with another character of the same type in digit and special character components. *Example*: $\mathbf{S_2}$ = **!!** can be transformed into **!#**.

4.3.2 Determining Password Changes

In this second approach, we have more information about the user's password habits: two old passwords. We can still take advantage of our first approach and use the most recent password to generate the EGrammar. However, we can also gather information about the changes made to the previous passwords and use this information in predicting the new password. We next discuss our algorithm to first determine the operations made to change the password, and then how to predict the new password based on the information.

In order to determine the changes between two passwords we implement a function that finds the minimum edit distance by creating a distance matrix. The function also involves a backtracking algorithm that determines the operations made between two strings. We have

developed our function based on the Damerau-Levenshtein [52] algorithm. The Damerau-Levenshtein edit distance is a string metric between two strings *s* and *t* which counts the minimum number of operations needed to transform one string into the other. In this algorithm an operation is defined as an insertion, deletion, or substitution, or a transposition of two adjacent characters. The algorithm starts by filling a (distance) matrix A of size $n_1 \times n_2$, where n_1 is the length of the first string *s* and n_2 is the length of the second string *t*. The record value in A[i, j] is the measure for the distance between the initial substring s_i of *s* of length *i* and the initial substring t_j of *t* of length *j*. At the time of creating this matrix, we also capture the operations associated to each step and store it in another matrix. Later by backtracking this matrix, we find the operations needed to transform one string to the other. See Appendices E.1 and E.2 for the implementation of the edit distance and the backtracking algorithms.

Note that our edit distance function is different than a regular Damerau-Levenshtein edit distance as mentioned before. Therefore, our algorithm needs to cater to this. Our algorithm is a hierarchical algorithm which first finds the edit distance between the simple base structures. A simple base structure is the base structure of the password without considering the length of each component. For example the simple base structure of *alice123!* is LSD. The first level of the algorithm applies the distance function as well as the backtracking function on the simple base structures of the given passwords to determine any changes between these strings. It then reverts some of changes and applies the distance function and the backtracking function on the new strings. This algorithm then creates a context-free grammar called TGrammar (Target grammar) that represents the transformational changes between the two passwords. In the next section we give a more detailed description of both algorithms. See Appendix E for a pseudo-code implementation of the full algorithm.

4.3.2.1 Hierarchical Transformation Algorithm. Given two old subsequent passwords, in the first level we parse both passwords into their simple base structures. As an example, suppose we have a sequence of two old passwords such as: *alice123!$* and *12alice$!*. The simple base structures are LDS and DLS respectively. Then, by calling our edit distance algorithm for these two simple base structures, we can determine the differences in the base structures. The edit distance matrix is shown in Figure 4.4. The bottom right element of the matrix is the edit distance between these two strings.

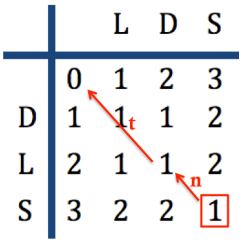


Fig. 4.4: The Edit Distance Matrix for Simple Base Structures (LDS and DLS)

Using the backtracking algorithm, we determine the operations that caused the change in the simple base structure. The backtracking algorithm starts from the bottom right corner of the matrix and travels back to the upper left corner of the matrix, and in each step determines what operation was done to calculate the edit distance. In this example, the function returns "tn" (**t**: transposition, **n**: no change) meaning that there has been a transposition in the first position, and no change in the next position. If a transposition is found in this step, we will transpose the components so that we neutralize the initial transposition effect and recreate one of the passwords similar to the other by applying the transposition. In the above example, the first password is changed into a new password by transposing the first two components creating *123alice!$*. This will count as one edit distance between these two passwords.

The second level of the hierarchical algorithm finds the edit distance between each component. We now use the changed password along with the second password to find the edit distance and the operations between these two strings.

As shown in Figure 4.5 the edit distance between *123alice!$* and *12alice$!* is 2 and the result of the backtracking function is "nndnnnnnt" (**n**: no change, **d**: deletion, **t**: transposition). We then count the edit distance of these two passwords as the sum of the edit distances of the first level and the second level hierarchy, which in this example is 3. Based on our defined operations, two adjacent components are transposed, '3' has been deleted and two adjacent symbols '!' and '$' are transposed. However, if we were to use the original Damerau-Levenshtein edit distance algorithm for these two strings without considering the hierarchy, the edit distance would be 6 (since a flat algorithm would find that there are 2 insertions in the beginning and 3 deletions and 1 transposition at the end of the string). Thus by developing the hierarchy algorithm and initially looking for any transposition in the simple base structures we can handle these situations better and create a more realistic edit distance function for passwords.

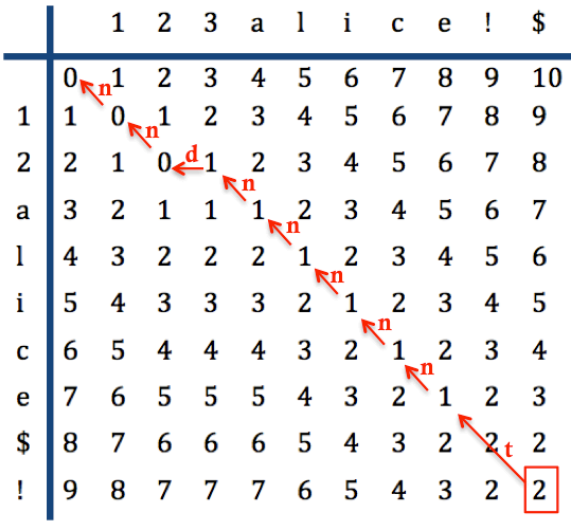


Fig. 4.5 The Edit Distance Matrix for Passwords (123alice!$ and 12alice$!)

In this system, users can also enter any names, numbers, important dates, and addresses as part of the information about the target into the appropriate boxes. For example, names of family members if known can be useful in password cracking since they can be used in substituting the alpha string components. Also, numbers such as date of birth, age, license number, social security number, etc. can be entered into the system. The numbers-related information will be added into the target grammar along with the information we capture from evaluating the transformations between the pair of old passwords.

4.3.2.2 Using Transformations to Create the Target Grammar. Once we learn what changes users have made to their old passwords, we can use the information to predict and guess their new password, with the hope and assumption that they apply similar modifications to their new password. In order to do so, we have developed an algorithm to generate new password guesses based on some of the most important changes we have found in our data and in the result of other studies [50, 51]:

1. Increment/decrement of the digit component by one: we recognize an increment or decrement by 1 in the digit component of the old passwords, and upon finding such alteration we add our prediction to our targeted grammar. For example if the old passwords are *bluemoon2* and *bluemoon3*, we would like to guess bluemoon4 with higher

probability value. Therefore, we add the same base structure L_8D_1, as well as 4 and bluemoon to our target grammar.

2. Insertion of the same digit: We have also developed algorithms to recognize if a digit has been inserted into a password and if it has been repeatedly added. Examples of such cases are: password → password5 → password55 → password555. In this case, for example, if the old passwords are bluemoon5 and bluemoon55, we add 555 to the target grammar as well as L_8D_3.
3. Capitalization of alpha strings: If the old passwords both have the same alpha sequence with different capitalizations, we add both of the capitalizations to our target grammar since the chances of using those masks are higher.

We convert the password predictions to a context-free grammar called TGrammar (target grammar). This grammar can be used to generate guesses that are variations of the prediction based on information obtained from user's old passwords. Both EGrammar and TGrammar are usually very small with only a few base structures that are only used to generate guesses very similar to the given passwords. In the next section we introduce a technique to merge two or more context-free grammars with different weights assigned to each grammar. Using this approach we can merge EGrammar and TGrammar with a more comprehensive grammar (or a general password cracking grammar) with higher weights assigned to EGrammar and TGrammar. This will allow us to generate a large number of guesses while giving higher priorities to guesses that are similar to the given input passwords.

4.3.3 Merging Two or More Context-free Grammars

Consider the scenario in which we have access to a single old password for a specific user. If our goal is to crack a new password created subsequent to the use of the old password, we would like to change the probabilities of the base structures and other components of the grammar in such a way that the guessing automatically generates passwords that are similar to the old password with much higher probability. However, we would also like to continue generating guesses like we would do normally in any password cracking since it is possible that the user has created a completely new and different password. This way, we maintain our guessing capability in highest probability order and we also guess the similar passwords (to the old password) earlier.

We define merging two grammars as follows:

Let G_1 and G_2 be two probabilistic context-free grammars. We define a new grammar G_3 called the *merge* of G_1 and G_2 and we represent it as:

$$\mathbf{G_3 = \alpha\, G_1 + (1- \alpha)\, G \quad where\ 0 \leq \alpha \leq 1}$$

Given a grammar rule R in G_1 or G_2, let the probability of R in G_1 be p_1 and the probability of R in G_2 be p_2. Then the probability p_3 of R in G_3 is:

$$\boldsymbol{p_3 = \alpha\, p_1 + (1 - \alpha)\, p_2}$$

Note that if R does not exist in one of the grammars, its probability is viewed as 0. The parameter α is used as a weighting factor between the grammars. Also note that after merging the probability values in each category also add up to 1, maintaining the properties of a probabilistic context-free grammar.

Table 4.2 Guesses Generated by MGrammar

pluto1995	pluto1993	pluto1915	pluto1495	1995Pluto	6pluto1995	pluto1995_
Pluto1995	pluto1992	pluto5995	pluto1895	1995plutO	pluto1234	_pluto1995
plutO1995	pluto1999	pluto1395	pluto7995	1995plUto	1q2w3e4r	pluto19951
plUto1995	pluto1996	pluto1195	pluto3995	1995pluTo	pluto!1995	pluto19955
pluTo1995	pluto1998	pluto1095	pluto6995	1995pLuto	123456	pluto17995
pLuto1995	pluto1965	pluto4995	pluto1995e	2pluto1995	pluto1995!	pluto01995
1995pluto	pluto1997	pluto1795	pluto1995r	3pluto1995	!pluto1995	pluto21995
1995	pluto1955	pluto9995	pluto1995s	4pluto1995	pluto@1995	pluto19953
pluto1985	pluto1945	pluto8995	qwerty	7pluto1995	pluto1995@	pluto16995
pluto1990	pluto1935	pluto0995	pluto1995E	0pluto1995	@pluto1995	pluto18995
pluto1975	pluto1925	pluto2995	pluto1995R	5pluto1995	pluto2008	pluto19925
pluto1991	pluto1295	pluto1695	pluto1995S	8pluto1995	pluto2009	pluto71995
pluto1994	pluto1905	pluto1595	1pluto1995	9pluto1995	pluto_1995	pluto19945

In this approach, every probability in the first Grammar G_1 will be multiplied with its weight **α** and every probability value in the second grammar G_2 will be multiplied with its weight **(1- α)**. Then if two similar rule values are exactly the same in both grammars, the probabilities are added together. The result is a special context-free grammar that can be used as before in offline attacks. Table 4.2 shows an example of password guesses generated with a merged grammar. The given password "pluto1995" was entered into the system and the resulted EGrammar was merged with a more general grammar. Different variations of pluto1995 can be seen among the first few guesses. Soon after, other password guesses such as "qwerty" or

"123456" are seen among the guesses since they are very common and have high probability values.

4.3.4 Testing and Result

As previously discussed, we did not initially have access to a dataset of sequences of changed passwords. However, we were able to obtain two such sets: (1) we were able to obtain a small list of 30 old and new passwords through a private party; and (2) we gathered 56 pair of old and new passwords through our survey study explained in section 4.2. We used the first set to learn how users change their passwords and to develop our system as explained in section 4.3.2.2. In this section, we present the result of our targeted password cracking system on the second set. In our survey, after the users changed their passwords, we asked the question: *Did you create your new password by slightly changing your old password for this website?* This question is important since we know which passwords were changed intentionally by slightly modifying the old password. Out of 56 pair of passwords obtained in this survey, 23 were claimed to be changed in this way. Therefore, in this section we only focus on those. We analyze whether we can crack/guess these passwords effectively. We input the old password to the system, and our goal is to crack/guess the new password early on during the guessing process. The system generates the EGrammar as discussed above. The EGrammar, in which we generate guesses within one edit distance of the given password, is useful most of the times. However, if the new password is changed considerably, it is more useful to merge the EGrammar with a more comprehensive grammar as discussed in section 4.3.3.

Table 4.3 shows the old password given to the system, the new password we try to guess, the number of guesses we made to find the new password using the targeted grammar, and the number of guesses we made to find the new password using our regular grammar. We also show whether we used the Edit distance grammar (Egrammar) or the merged grammar (Mgrammar) in our targeted attack. We used Yahoo-train and our NPC system to construct the grammar. We also used dic0294 as our attack dictionary. We limited the number of guesses to 10 billion guesses in our password cracking sessions. The results show that we were able to guess most of the passwords that were changed slightly. The reason we were not able to crack some of the passwords was mostly due to not having the alpha string part in our attack dictionary. The result also shows that only a few of the passwords were broken during a normal password cracking

attack within 10 billion guesses made. The targeted attack was more efficient when information about old passwords of users were available.

Table 4.3 Test Result of Targeted Attack

Old password	New Password	Number of Guesses in Targeted Attack	Number of Guesses in Regular Attack	Grammar
tharaborithor	thorborithara	--	--	--
Simba144!	@Simba2523	734,505,973	--	MGrammar
$unGl@$$220	$unGl@$$110	4,070	--	MGrammar
research!	Research!	554	5,059,949,503	EGrammar
starWars@123	star#Ecit@123	2,227,558	--	EGrammar
thebigblackdogju	blackdogmoretim	--	--	--
Ahk@1453	Ahk#1453	12,026	--	EGrammar
qpalzm73	qpalzm73*	1,810	--	EGrammar
pluto1995	boonepluto	--	--	--
caramba10	caramba12	14	11,424,542	MGrammar
Elvis1993!	Professional1993	--	--	--
pepper88	peppergator88	128,197,109	2,563,504,751	MGrammar
ganxiedajiA1!!	1ganxiedajiA	7,794	--	MGrammar
88dolphins!	55dolphins!	38,503	--	MGrammar
kannj2013!	kannj2013	97	--	EGrammar
!FSU$qr335	!FSU$qr335mcdd	--	--	--
vballgrl77	schatzimae	--	--	--
nickc1007	corkn1007	--	--	--
sunflower12	sunflower13	202	119,336,969	EGrammar
meg51899	Meg51899*	5,381	--	EGrammar
Research1	research11	206	23,728,452	EGrammar
Gleek1993	Gleek1985	9,661	1,994,709,669	MGrammar
Oaklea0441	Oaklea0112	91,014	--	MGrammar

CHAPTER 5

PASSWORD CHECKING/STRENGHTENING

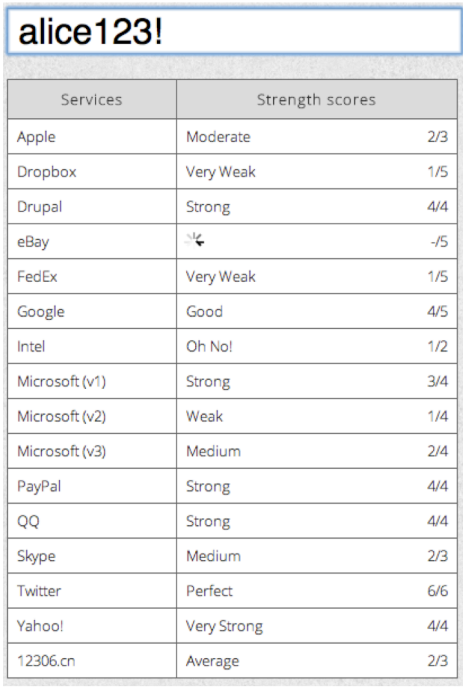
In this chapter I turn to the importance of passwords for security and protecting information for users. I investigate the question of how to measure password strength and how to help users create stronger passwords. I first discuss previous work on password meters and password strengthening. I then discuss other techniques used to make passwords stronger such as rule-based approaches. I then review the AMP password analyzer and modifier [5] which was first introduced in my Master's thesis. I later explore this approach further and analyze its effectiveness in detecting weak and strong passwords and suggesting stronger passwords with slight modifications. Part of the work in this chapter appeared in [53].

5.1 Background and Motivation

When it comes to password security, the main concern is that people do not have enough knowledge about what a strong password is and how to create one. Most organizations and websites follow a rule-based approach in recommending or enforcing password policies. Their aim is to help users create a stronger password. Password policies have certain rules such as "your password must contain at least two digits", or "your password must be at least 8 characters long". Some other websites have recommendations and use password meters to show the strength of the user selected password. A study by Shay et al [49] was conducted to seek an understanding of the factors that make creating and following password policies difficult. They gathered the results of a survey of 470 Carnegie Mellon University students, faculty and staff. Their results imply that users were not happy about changing the password creation policy to a stricter one and they also found that about 80% of users reused their passwords across different accounts and 60% used one password with slight changes for different accounts. Riley [54] also found that the average length of time users maintained their primary personal use password was reported as 31 months and 52% of users never change their password. These studies show that having an effective password creation policy does not always mean having strong passwords and a secure system, since users are forced to create passwords that may not be easy to memorize

(which is not good), and most users tend not to change their passwords often nor do they have different passwords for different websites.

Rule-based advice is confusing as there is no consistency across websites in the requirements, with differing advice about length, number of symbols and digits, and even in the symbols that are allowed. In [55] it is shown that inconsistent and even contradictory recommendations make such advice unreliable for users. A recent study [56] analyzed password meters in popular websites and shed light on inconsistencies in determining the strength of passwords across different platforms. The authors created a system that allows a use to enter a password and checks the strength of the password based on different websites. Figure 5.1 shows an example of the result of password strength meters against password "alice123!". [57] reports that although nowadays users understand the importance of secure behavior, they still find it too difficult to cope with password creation policies, and they rarely change their passwords due to the frustration of creating a new password along with the difficulty of memorizing it. In studies by Charoen et al. [58] and Adams and Sasse [59], it was found that users are not even unanimous about the necessity of having a strong password and the reason users choose insecure passwords is because they usually do not know how to create secure ones.


alice123!

Services	Strength scores	
Apple	Moderate	2/3
Dropbox	Very Weak	1/5
Drupal	Strong	4/4
eBay		-/5
FedEx	Very Weak	1/5
Google	Good	4/5
Intel	Oh No!	1/2
Microsoft (v1)	Strong	3/4
Microsoft (v2)	Weak	1/4
Microsoft (v3)	Medium	2/4
PayPal	Strong	4/4
QQ	Strong	4/4
Skype	Medium	2/3
Twitter	Perfect	6/6
Yahoo!	Very Strong	4/4
12306.cn	Average	2/3

Fig. 5.1 Example of Inconsistencies across Different Password Meters

The U.S. NIST guideline [2], the basis for most rule-based policies, proposed a rule-based approach that used the notion of Shannon entropy for estimating password strength based on suggested values of the components of the password. However, researchers [3, 4, 27] showed that the use of Shannon entropy as defined in NIST is not an effective metric for gauging password strength. Weir et al. [3] performed password cracking attacks against multiple sets of real life passwords and showed that the use of Shannon entropy as defined in NIST does not give a sufficient model to decide on the strength of a given password. Castelluccia et al. [27] also perform studies and showed that insecure passwords are accepted and secure passwords are rejected as a result of this approach.

In the next section we discuss our approach which is based on an analyze-modify approach in which we first estimate the strength of a password based on real cracking attacks and then modify a weak password to create a strong one for the user within an edit distance of one. At the time of this research there were only a few relevant studies that are similar to our approach in some ways. Schechter et al. [60] proposed to build an oracle for existing passwords that are available to the Internet-scale authentication systems. The authors recommend that popular passwords be disallowed and the main thrust of their work is to devise a way to efficiently store the large number of popular passwords that would be prohibited. They use the notion of a count-min sketch (similar to a Bloom Filter) for such storage. Their proposed oracle would disallow very popular passwords while otherwise allowing users to choose any password they wish. An open question posed in their study is how to use the oracle without revealing the actual password to attackers while querying online. Our technique gets around this problem as well as their storage problem. Castelluccia et al. [27] explored measuring the strength of passwords using a Markov approach. They spent a fair amount of their study proving the security of their system; however, they did not show the effectiveness of the Markov approach in estimating the strength of passwords against real attacks.

5.2 Analyzing and Modifying Passwords

In this section we review our work on estimating password strength and creating stronger passwords [53] and the system we developed called AMP. The key to a good password checker is the ability to help a user create a secure password while ensuring the password is easy for the particular user to memorize. Both of these aspects are important since it is very easy to develop a

policy that results in strong passwords (using random password generators) that are particularly unusable. In our approach we use an implicit password creation policy in which there is a reject function that rejects a weak password and then a modify function that changes the weak password slightly to one which is appropriately strong.

For a password to be strong we need to make sure that it cannot be easily broken. The first step in AMP [53] is to evaluate the user chosen password for strength. We define the password strength as the probability of the password being cracked by an attacker. We take advantage of the probabilistic context-free grammar (discussed in chapter 3) trained on a set of real user passwords to estimate the probability of a password being cracked. We assume that this set is a comprehensive set of passwords (and a sufficiently large sample set) that can be used as a model of realistic passwords. In fact, we are able to determine a *threshold value* below which a password would be considered as strong. This allows us to build a *reject function* that accepts a strong password and rejects a weak one. AMP then modifies weak passwords to ones that are strong but within edit distance of one.

5.2.1 Setting the Threshold

A strong password is one for which it takes an attacker an appropriately long *cracking time* (*ct*) to crack that password (in hours). In an optimal attack, the attacker would try different guesses in decreasing order of probability. We define the ***threshold*** (*thp*) as a probability value such that passwords with probability less than *thp* are strong and those that are greater than or equal to *thp* are weak. By using the probabilistic context-free grammar (plus appropriate dictionaries) as our model of the realistic password distribution, we can determine the number of guesses *g(thp)* the attacker would make before trying a password with a value equal to the threshold value *thp*. Let *r* be the rate-per-hour of the guesses (based on the hash type, cracking system speed, etc.). We thus have g(*thp*) = *ct* * *r*. In the preprocessing phase, we create a table that contains the number of guesses and the probability values of the guesses at various time intervals by running the probabilistic password cracker. This is then used as a mapping of *thp* to g(*thp*) and is used to select what level of security we want the system to have. The threshold is then used to decide whether a given password is strong or weak.

The AMP system first asks users to enter their chosen password; the probability of the chosen password is then calculated using a probabilistic context-free grammar. The password is

parsed to its base structure and components. For example "*Alice123!*" will be represented as $L_5D_3S_1$. Then the probability of the base structure $L_5D_3S_1$ along with the probabilities of *alice*, *123*, *!* are found from the grammar. The product of these probabilities is the probability of the user's password. This probability p_u is compared with the threshold value to accept or reject the password. If p_u is smaller than the threshold value, the password is strong, meaning that it will take longer to guess the password using an optimal attack.

5.2.2 Modifying a Weak Password

When a password is weak and is rejected by the system, the system then tries to modify it slightly to create a stronger password for the user. The modification needs to be minor since we would like to keep the password usable and memorable. A usable password is a password that is easy for the user to remember and type. Things people can remember are different for each group of people based on their age, situation, location, etc. If a password is weak we try to create passwords with slight changes to the user-chosen password using the AMP distance function. This is based on Levenshtein edit distance to fulfill the need of usability for users. We believe users choose password components for memorability and only minimal changes should be made. Hence, we start generating passwords with distance one from the user-chosen password and check if the modified password is within the acceptable threshold value. If we find one, we are done and the new password is recommended to the user, otherwise we continue and check all possible changes. Obviously, it is possible that we might not be able to create a password within distance one with the desired probability value.

5.2.3 Updating the Grammar

In order to maintain the AMP system as still effective after users use the system for some time, an update strategy is developed that modifies the grammar periodically. After using the system for a period of time the probability distribution of passwords changes. Since the supposedly strong passwords suggested by AMP have become in use more often and would now have higher probability in the guessing generator, the attacker has a better model of the AMP generator and therefore continued use of the original grammar could be problematic. Therefore, every modified password that has been suggested to a user is considered as a publicly disclosed password. Using an appropriate weight, the password can be added to the training set effectively.

This ensures having a realistic and up to date probability distribution for the probabilistic context-free grammar at all times. In order to update the training set, there is no need for processing the training set again; we only need to adjust the probability values in the context-free grammar and it can be done almost instantaneously. We have shown that the Shannon entropy of the grammar seems to be approaching the theoretical maximum Shannon entropy as we update the grammar. We also found a similar result for the guessing entropy. Theoretically, having a uniform distribution for passwords is ideal since all passwords will have equal probabilities. Practically, this would mean that each password is equivalent to being randomly chosen. Note that using our update algorithm we are moving closer to a uniform distribution but are likely very far away from it.

5.3 Testing and Result

In this section, I discuss the result of our analysis of the effectiveness of the AMP system on several revealed password sets. We randomly created three different sets for (1) training the AMP password checker (RockYou: 1 Million, MySpace: 30,997, Hotmail: 4874); (2) testing the AMP system (RockYou: ½ Million, MySpace: 15,499, Hotmail: 2,437); and (3) training a probabilistic password cracker (RockYou: ½ Million, MySpace: 15,499, Hotmail: 2,437).

We used the first set as the training set to construct a context-free grammar for the AMP password checker. We then set the threshold value to one day, equivalent of 43.2 billion guesses for our experiments. Therefore, a password is called weak if it can be cracked/guessed within one day, and it is strong otherwise. We then give the second set (test set) as the input to the AMP system. The system calculates the probability of each password in the set and compares it against the threshold. If the password is weak, the system tries to generate a strengthened password within edit distance of one.

We can categorize the result of the AMP system into four different groups: (1) Passwords determined as strong, (2) Passwords determined as weak and the system was not able to strengthen them, (3) Passwords determined as weak and the system was able to strengthen them, and (4) The strengthened (modified) passwords from the third category. We ran a series of password cracking sessions using two different password crackers (the probabilistic password cracker and John the Ripper). The results of the password cracking are shown in Table 5.1 and 5.2.

Table 5.1 Password Cracking Results using John the Ripper

	(1) Originally **Strong** Passwords	(2) Originally **Weak** Passwords (not able to strengthen)	(3) Originally **Weak** Passwords (able to strengthen)	(4) **Strengthened** Passwords
Hotmail				
$\frac{cracked}{total}$	$\frac{2}{325}$	$\frac{49}{53}$	$\frac{988}{2,059}$	$\frac{2}{2,059}$
Percentage	(0.61%)	(92.45%)	(47.98%)	(0.0975%)
MySpace				
$\frac{cracked}{total}$	$\frac{23}{1484}$	$\frac{104}{149}$	$\frac{5,343}{13,866}$	$\frac{71}{13,866}$
Percentage	(1.55%)	(69.80%)	(38.53%)	(0.51%)
RockYou				
$\frac{cracked}{total}$	$\frac{281}{32,794}$	$\frac{22,248}{24,745}$	$\frac{235,302}{442,461}$	$\frac{1,186}{442,461}$
Percentage	(0.86%)	(89.90%)	(53.18%)	(0.27%)

Table 5.2 Password Cracking Results using Probabilistic Password Cracker (PPC)

	(1) Originally **Strong** Passwords	(2) Originally **Weak** Passwords (not able to strengthen)	(3) Originally **Weak** Passwords (able to strengthen)	(4) **Strengthened** Passwords
Hotmail				
$\frac{cracked}{total}$	$\frac{1}{325}$	$\frac{53}{53}$	$\frac{1,069}{2,059}$	$\frac{113}{2,059}$
Percentage	(0.3%)	(100%)	(51.91%)	(5.48%)
MySpace				
$\frac{cracked}{total}$	$\frac{27}{1,484}$	$\frac{135}{149}$	$\frac{8,341}{13,866}$	$\frac{698}{13,866}$
Percentage	(1.81%)	(90.60%)	(60.15%)	(5.03%)
RockYou				
$\frac{cracked}{total}$	$\frac{467}{32,794}$	$\frac{24,378}{24,745}$	$\frac{259,027}{442,461}$	$\frac{18,134}{442,461}$
Percentage	(1.42%)	(98.51%)	(58.54%)	(4.1%)

The results show that both originally strong and strengthened passwords (modified from weak passwords) have very low rate of cracking compared to weak passwords. John the Ripper

generally cracked less than 1.5% of the strong passwords and the Probabilistic Password Cracker cracked about 5%. Overall, without using the AMP system the total rate of cracking the test set (columns 1,2,3) was 56.6% with the probabilistic password cracker. Using AMP and not allowing weak passwords to be selected by users, the cracking rate is 3.9%. The AMP system successfully distinguishes weak passwords from strong ones with an error rate of 1.43% (column 1). This rate is the percentage of passwords originally identified as strong, but that can be cracked.

Besides using the 1-day threshold, we also ran similar tests as the above using threshold values for 12 hours, 48 hours and 96 hours. Figure 5.2 shows the total rate of cracking the test set before using AMP and after using AMP for both John the Ripper (JTR) and the probabilistic cracker (PPC). The time allocated for cracking was of course the same time as used for determining the threshold. Note that the results are similar to the 1-day results and even at 4 days we are significantly improving the weak passwords.

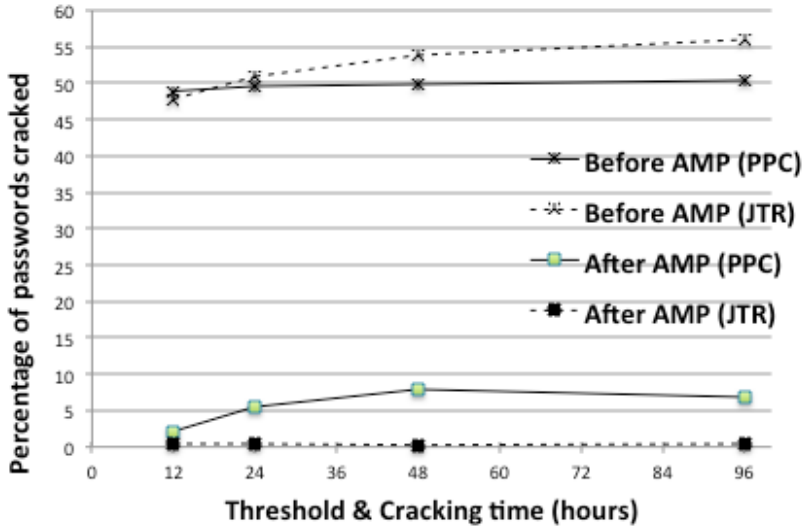


Fig. 5.2 Using AMP for Different Time Thresholds

CHAPTER 6

IDENTIFING PASSWORDS ON DISK

As discussed previously, with the increase in the number of accounts and passwords that each user has, and the recommendations on not reusing passwords, users are faced with the problem of how to create secure and memorable passwords. Thus, they are increasingly turning to saving their passwords in some manner, either on paper or on their computers. A survey in 2012 by Kaspersky Lab [61] revealed that 29% of users store their passwords on media. 13% create a text document on the hard drive, 9% write them on a cell phone, and only 7% use specialized software. In our own recent informal survey of 100 students, we found that 42% store their passwords and 55% of these do so on disk or cell phone in clear text without encryption or using specialized software. As password policies are becoming more complex, we believe that users turn more to storing them on media. In this chapter I describe work done that gives a solution to the problem of *identifying passwords on media* [62].

Suppose that law enforcement captures a hard disk and needs to find if there are any stored passwords on the disk. An example scenario [63] is when there are encrypted files on disk (say illegal photos). It is possible that the user has stored the password somewhere on the disk to easily access these encrypted files. An investigator could look at each file and try to determine by context and structure which strings might be passwords. This would of course be quite tedious, especially with very large disks. Investigators sometimes use existing tools to tokenize all the strings on the disk and use these as a dictionary for offline cracking of the encrypted files. The list, however, often becomes too large to build a dictionary that can be subsequently used for cracking. The *identification* problem is to distinguish the tokens that are more likely to be passwords and winnow down the list to a more manageable one. This problem is non-trivial because distinguishing which of the strings are passwords from a large set of strings has no obvious solution.

There have been a few studies on how to find passwords used for encryption or passwords stored through browsers [64, 65, 66]. However, we are unaware of any work that tries to distinguish passwords from other strings that are stored directly by the user on disk. Garfinkel et al. [67] discuss a more general problem of trying to capture a storage profile for each computer

to detect anomalous behavior. They propose to monitor the forensic content of the disk (or other media) such as email addresses, credit card numbers, etc. There exist recovery tools such as EnCase [68] and FTK [20] that have the capability of finding strings on disks. However, the real problems is filtering these strings and determining the likeliest strings, which might be passwords. Identity Finder [69] is also a commercial sensitive data manager toolkit. It looks for passwords, credit card numbers, social security numbers, etc. in a variety of places including files, emails, browsers, registry files, databases, and websites. Furthermore, Identity Finder provides password search customization by enabling certain keyword and regular expression search thus requiring the investigator to propose a string search. Again, this tool does not tackle the password identification problem.

In this work, we first analyze a disk image and retrieve all strings on the disk that could possibly be passwords, which we call *tokens*. During this process we prune a potentially very large set of tokens to a more manageable set that we expect contains most of the passwords. We then use the probabilistic context-free grammar discussed in chapter 3 to calculate the probability of each token and decide which ones are likely to be passwords. In the final phase we use these probabilities and develop a set of ranking algorithms to suggest an ordered list of tokens. This list can be then used as a dictionary by the investigator to do password cracking using any appropriate approach.

6.1 Retrieving Tokens from the Disk

In this work the assumption is that a user is simply storing the passwords in a file on disk in order to remember their password if needed. The file can be in allocated space or unallocated space (file might be deleted) or hidden through the operating system. The first step is to retrieve all the files from the disk image. We use tsk_recover tool to recover files from both allocated and unallocated spaces. Tsk_recover is part of an open source digital forensics tool for analyzing disk images called Sleuth Kit [70]. We then consider file types that are more probable to contain text and to be modified by users such as .doc, .docx, .xls, .xlsx, .rtf, .odt, .pdf, and .txt. We use open source tools such as *catdoc*, *docx2txt*, *xls2txt*, *unoconv* and *xls2txt*, *unrtf*, *odt2txt*, and *pdftotext* to convert these files to text file format to be able to read the contents of the files. We then tokenize the strings for each file using space, tab and newline as delimiters. We store tokens of each file in a new file associated to the original where each token is written on a single line. We later search

for possible passwords through these associated files. Even an average sized disk typically contains many different file types and files with text content resulting in a huge number of tokens. In order to be able to reduce the set of tokens that we retrieve, we define a set of rules that filters out some classes of tokens that we believe are very unlikely to be passwords. We developed two different sets of filters. Initial filters are developed to eliminate tokens with characters, or lengths that are not part of passwords. Specialized alpha string filters are developed to reduce the number of alpha strings in the text files.

6.1.1 Initial Filters

We examined some revealed password sets to get insight into what kinds of structures are rarely seen in passwords. The set of initial filters that we define and apply is as follows:

- **Non printing**: These are ASCII characters that are almost always not valid password characters.
- **Length**: Passwords usually have certain lengths based on the policies enforced on each website. Here we apply a conservative bound and only consider the tokens with length l, $6 < l < 21$.
- **Floating point**: The files on disk (especially the .xls files) can often include many floating point numbers. We filter out all floating point numbers since our studies on revealed password sets show that there is very little chance of such tokens being real passwords. We therefore filtered out, using a regular expression, any string of the form [-+]? [0-9]* .? [0-9]+ ([eE][-+]?[0-9]+)?
- **Repeated tokens**: In each file, we only keep one copy of tokens that are repeated multiple times. One might think that repeated tokens are not likely to be passwords, but it is possible that users store password information for many different accounts and thus would have multiple copies in a file.
- **Word punctuations**: We remove tokens that seem to include punctuation patterns of a sentence by filtering out tokens that contain only alpha strings *ending with* any of the following characters: ;:,.?!-)}. We also filter out such tokens *starting with* (or {. Our examination showed that only 0.516% of such tokens are found in a sample of 1 million passwords in the Rockyou set.

6.1.2 Specialized Alpha String Filters

An extremely prevalent class of tokens found on a hard disk is the set of alpha strings (those containing only alphabetic characters). In this section we describe various approaches to handling such strings. We define the specialized alpha string filters as follows:

- **All-alphas**: This filter eliminates tokens that are all alpha strings. In this case we assume that most of the time passwords do not contain only alphabet characters but also contain digits or special symbols as well. This is further enforced by current password creation policies.
- **Sentences**: This filter tries to eliminate all alpha strings that are part of sentences. To detect sentences we use OpenNLP [71]. This tool can detect whether a punctuation character marks the end of a sentence or not. It cannot however identify sentence boundaries based on the contents of the sentence. An additional problem we faced was that during the conversion process to a .txt file, word wrapping is not preserved as line breaks are added, so sentences, which continue into another line are considered separate indices by OpenNLP. We thus verify if an index starts with a capital letter and ends with a period and filter out such sentences.
- **Capitalization**: This filter eliminates all lower case alpha strings. This is because some of the password policies allow you to have passwords, which contain one or more of the classes (symbols, digits, and capital letters).
- **Dictionary words**: This filter eliminates alpha strings that appear in a dictionary. The purpose of using an English dictionary is to try to eliminate words that are most likely part of sentences in the documents, and keep the rest of the strings in our token set.
- **Multiwords**: This filter eliminates all alpha strings that are not multiwords. Examples of such strings are passphrases (without whitespace) that appear to be increasingly used as passwords.

6.2 Identifying Passwords

After examining the disk and retrieving all tokens separated by whitespace, our main task is to distinguish and find passwords from other sequences of characters that appear in a text file. For this purpose we use the probabilistic context-free grammar trained on a large set of real user passwords. As discussed before, this grammar models a password distribution and the way users

create the passwords. This helps in differentiating passwords from regular text. As explained previously in section 5.2, given a probabilistic context-free grammar we can calculate the probability of a given string in the password distribution. We parse the given string into its components and find the probabilities associated with each component from the grammar. We then calculate the probabilities of all of the retrieved tokens remained after applying the filters.

6.2.1 Ranking Algorithms

After retrieving all tokens and calculating the probability value of each token, we rank the tokens in order to output a limited set of tokens (say the top *N* tokens) for the investigator to examine as the most likely possible passwords from the hard disk. Obviously, the ideal is having both high precision and high recall in this potential password set. Recall can be more important in an offline attack while precision might be more important in an online attack. We believe that it is very important to reduce the size of the potential password set even in the case of offline password cracking; although computers have become much more powerful through the use of GPUs etc., many hashing algorithms (for example the one used in TrueCrypt) can still purposely take a very long time for the resources available to typical law enforcement.

In this section we discuss our three different algorithms for ranking the possible passwords. Recall that we maintain the associated files the tokens belong to and we use this relationship in our algorithms. We use a parameter *N* that is the number of potential passwords that we return to the investigator. The three different approaches that we evaluated are:

Top Overall: In this natural approach we select the *N* highest probability tokens from all of the retrieved tokens.

Top Percent (per File): In this approach, we select an equal percentage of the highest probability tokens from each file such that the total number of tokens returned is *N*. The resulting tokens are then ordered by their probabilities.

Top 1-by-1 (per File): In the first round, we choose the highest probability token from each file and then rank them by highest probability. In the second round we select the second highest probability token from each file (if available) and again rank them by highest probability. We repeat this until we reach the desired *N* tokens. Note that tokens from round *j* are ranked above round $j + 1$.

6.3 Testing and Result

In this section we discuss our test results both on the utility of our filtering techniques as well as the effectiveness of our algorithms to identify passwords. We used the Digital Corpora Govdocs1 [72] as the source of our files to create test disks. This corpus contains about one million freely redistributable files in many different file formats. We then added real user passwords taken from revealed sets of passwords to the files since we did not have access to test disks that contain known real passwords. For our testing purpose we created five data disk images of different sizes. See Table 6.1. For our disks we only used files likely to be created by the user (.doc, .xls, .pdf, etc.). The sizes of the data disk images in Table 1 are therefore only the total data sizes of these files.

Table 6.1 Test Disk Images

Data Disk Image Size	#Files Analyzed
1 GB	1194
500 MB	571
250 MB	426
100 MB	143
50 MB	108

Table 6.2 Reduction of Tokens due to All Filters

	50 MB	100 MB	250 MB	500 MB	1 GB
# Before filtering (millions)	2.45	2.16	6.76	28.84	49.41
# After filtering (millions)	0.07	0.050	0.25	1.38	3.21
Total reduction (percent)	97.15	97.68	96.35	95.21	93.50

We randomly selected passwords from revealed password sets and then randomly selected a file to which to add each password. The result of our filtering shows that all of the filters except Non-printing have a major impact on the end result, reducing the large number of tokens we obtain from the hard disk to a much smaller set. The Non-printing filter is important in our next step of calculating the probabilities but was rarely actually useful for reduction. In the 1 GB disk, the length filter reduced 53% of tokens, the floating point filter reduced about 28%, the

repeated token reduced 70%, the word punctuation reduced 20% and the all-alphas reduced 33% of tokens. Table 6.2 shows the number of tokens (in millions) before and after filtering and the percentage of reduction when applying all of the filters.

6.3.1 Testing Ranking Algorithms

In this section we explore result of our test on the ranking algorithms. We used two revealed sets, Rockyou and CSDN, from which we chose passwords to store on the disks. We applied the initial filters and the all-alphas filter. We stored 5 passwords on each disk in one series of test and 15 passwords in a second series of tests. We believe this represents a range of passwords that a normal user might have stored. We used Yahoo-train for training the probabilistic context-free grammar that is used to calculate the probabilities of the potential passwords. We then determined how many passwords we are able to find by each of the algorithms. We determined the results when returning *N* potential passwords to the investigator, where *N* is 1000, 2000, 4000, 8000, and 16000. In the following tables we show the number of passwords found in the disk by the algorithms (true positives). In Table 6.3 we show the results for storing 5 passwords from CSDN.

Table 6.3 Number of Found Passwords (Out of 5 from CSDN)

		50 MB	100	250	500	1 GB
N=1000	Top overall	1	2	0	0	2
	Top percent	2	3	1	1	2
	Top 1-by-	**5**	**3**	**2**	**3**	**3**
N=2000	Top overall	1	2	0	0	2
	Top percent	5	3	1	1	2
	Top 1-by-1	**5**	**4**	**2**	**3**	**4**
N=4000	Top overall	5	2	0	0	2
	Top percent	5	3	2	1	2
	Top 1-by-1	**5**	**5**	**3**	**4**	**4**
N=8000	Top overall	5	3	0	0	2
	Top percent	5	3	2	1	3
	Top 1-by-1	**5**	**5**	**4**	**4**	**5**
N=16000	Top overall	5	4	0	0	2
	Top percent	5	4	2	3	3
	Top 1-by-1	**5**	**5**	**4**	**5**	**5**

As an example in Table 6.3, using the 1-by-1 algorithm we can find all 5 passwords on the 50 MB data disk, 3 passwords on the 100 MB data disk, 2 passwords on the 250 MB data disk, etc., within the top $N = 1000$ returned by the algorithm. When comparing algorithms given an N and the number of stored passwords, a higher recall implies a higher precision and they can be both calculated from the number of passwords found. For example, the average recall value of the 1-by-1 algorithm across different disk sizes for $N = 8000$ is 92%, for the top percent algorithm is 56% and for the top overall algorithm is 40%. This shows that the 1-by-1 algorithm has both higher precision and higher recall compared to the other algorithms.

In Table 6.4, we show the results for storing 15 passwords from CSDN. For $N = 8000$ the average recall value of the 1-by-1 algorithm is 89.3% across the different disk sizes. The results show that the 1-by-1 algorithm is quite good and better than the others. Results of storing passwords from the Rockyou password set showed similar results.

Overall, it appears that the 1-by-1 algorithm is consistently the best. Recall that in our experiments so far, the filters eliminated all alpha strings. We believe that this is reasonable, as today's password policies would almost invariably disallow such passwords. However, we next explore whether less restrictive filtering of alpha strings can be useful.

Table 6.4 Number of Found Passwords (Out of 15 from CSDN)

		50 MB	100	250	500	1 GB
N=1000	Top overall	1	7	0	2	2
	Top percent	4	10	2	3	3
	Top 1-by-1	**11**	**12**	**7**	**8**	**9**
N=2000	Top overall	1	9	0	2	2
	Top percent	9	10	2	4	5
	Top 1-by-1	**12**	**14**	**9**	**9**	**11**
N=4000	Top overall	11	10	0	2	2
	Top percent	10	11	3	5	6
	Top 1-by-1	**15**	**15**	**12**	**10**	**12**
N=8000	Top overall	13	11	0	2	2
	Top percent	11	11	8	5	8
	Top 1-by-1	**15**	**15**	**13**	**10**	**14**
N=16000	Top overall	15	14	0	2	2
	Top percent	12	14	9	8	8
	Top 1-by-1	**15**	**15**	**13**	**11**	**14**

6.3.2 Testing Specialized Filtering

In this section we compare the specialized filters as defined in section 6.1.2. The specialized filters are applied in addition to the initial filters. We used the 1 GB disk and stored 15 passwords from the Rockyou set. The results are shown in Table 6.5. The numbers in parenthesis show how many of the 15 passwords stored on the disk remained after the filtering process. For example *All-alphas (11)* shows that four of the passwords stored on the disk were filtered out due to the filtering process. The results show that using less aggressive filters such as multiwords or dictionary words does reduce the password loss due to filtering. However, these approaches are not as successful as the more aggressive approach (all-alphas filter) in subsequently identifying the passwords because they still retain too many alpha strings.

Table 6.5 Number of Found Passwords (Out of 15 from Rockyou)

		No Filter (15)	Caps (11)	Multi words (14)	Dictionary (14)	Sentences (15)	All-alphas (11)
N=1000	Top overall	0	2	0	0	0	5
	Top percent	1	1	3	3	2	1
	Top 1-by-1	**2**	**2**	**4**	**4**	**0**	**8**
N=2000	Top overall	0	2	0	0	0	5
	Top percent	1	2	3	3	2	2
	Top 1-by-1	**2**	**2**	**4**	**5**	**0**	**10**
N=4000	Top overall	0	2	0	0	0	5
	Top percent	2	3	3	3	3	4
	Top 1-by-1	**2**	**2**	**5**	**5**	**1**	**10**
N=8000	Top overall	0	2	0	0	0	5
	Top percent	4	4	5	5	3	7
	Top 1-by-1	**2**	**2**	**7**	**7**	**1**	**10**
N=16000	Top overall	0	2	0	0	0	5
	Top percent	4	4	5	5	3	7
	Top 1-by-1	**4**	**5**	**8**	**8**	**7**	**10**

When applying the multiwords filter, we keep a more limited set of alpha strings compared with the dictionary filter. The multiwords filter eliminates all single words (whether it is a dictionary word or not), whereas the dictionary filter eliminates only the single words that

are included in the dictionary and still keeps the multiwords. For the dictionary filter we used the training dictionary that we created in section 3.1.2.

We observed in the tests that we usually end up having a large number of alpha strings with fairly high probabilities because of the way the probability of each token is calculated (having equal probability values for all the words of the same length). Therefore, when the top *N* potential passwords are selected, we do not find as many of the passwords as quickly as we could if we eliminated all of the alpha strings.

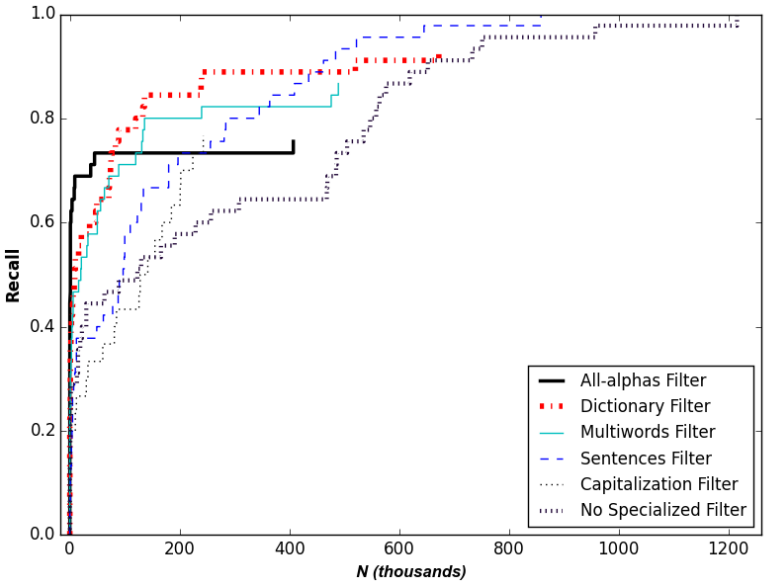


Fig. 6.1 Comparison of Specialized Filters as *N* Varies

We explored the 1-by-1 algorithm in more details using the various specialized alpha string filters. In Figure 6.1 we plot *N* versus the recall value for all integer values of *N* until we find all of the passwords that can be found by that filter (the results are averages of several runs). The aggressive all-alphas filter may not be able to find all of the passwords, but on average finds 9 of the passwords (recall of 0.6 and precision of 0.005) at $N = 1{,}659$. In comparison, when applying no specialized filter we find 9 of the passwords only at $N = 229{,}671$.

The results show that applying our filtering and identification approach allows an investigator to find most of the passwords within a very reasonably small value of *N* and avoid having to check a huge number of strings. Note that if the aggressive filter is not successful for the investigator's needs, a less aggressive filter can then be tried. For example, the dictionary

filter which loses fewer passwords finds on average 9 of the passwords at $N = 36{,}240$ which is still much better than no specialized filtering. By choosing the appropriate value of N, the investigator can move between online and offline attack situations. The all-alphas filter identifies about half of the stored passwords even within the first 500 proposed tokens. Sometimes even finding one password on the disk could be very helpful for an investigator since users typically use the same password for many sites / accounts. For the all-alphas filter the first password was found on average at $N = 11$.

CHAPTER 7

CONCLUSION

The main focus in this disseration is creating more efficient password crackers using probabilistic context-free grammars. I have shown how to learn new patterns (such as keyboard and multiwords) when creating grammars for password cracking. I have shown how to handle several issues that arise from adding more complicated patterns to the grammar. I also developed metrics to be used for comparing, analyzing and improving attack dictionaries. The results have shown that the addition of such patterns can significantly improve the password cracking (55% improvement over previous work). As an example, this can be interpreted as taking 2 seconds to have a 50% chance of cracking a single password versus taking 26 minutes using the previous system. The techniques described in this dissertation can be used in developing other patterns and can be applied to other password crackers. I have also introduced targeted attack, in which several grammars are created based on available information about a target such as names, numbers, or previous passwords. The grammars are then combined together to create a grammar with higher probability values assigned to more relevant components and values.

An immediate use of this work is in creating an efficient password cracker that can help law enforcement in cracking passwords for accounts or password protected files for their cases. However, learning password distribution and how users create passwords can help in other areas as well. In this dissertation I have shown several applications of using the probabilistic context-free grammars for passwords. The AMP system is described that estimates password strength against real attacks and helps users create stronger passwords. I have presented results of real password cracking sessions to confirm the accuracy of this metric.

Another application of the probabilistic context-free grammar technique is also introduced in which passwords stored on media are discovered. I have shown how to retrieve a small set of possible passwords from a large disk by applying filtering techniques and ranking algorithms. The results show that by returning a set of 2000 tokens, 60% of passwords can be successfully identified. In future work, the system can be adapted to identify passwords on cellphones and USB drives. Also, more filtering techniques can be developed to find passwords for targeted individuals.

APPENDIX A

PSEUDO-CODE FOR MULTIWORD SEGMENTATION

```
procedure FINDALLMULTIWORDS(word, n)
    ret = [] //list of potential breakdowns
    if  n==1  then
        return
    end if
    for (i=0; i< word.length ; i++) do
        left = word.substring(0,i)
        if ISDICTIONARYWORD(left) then
            right = s.substring(i, word.length)
            if   ISDICTIONARYWORD (right) then
                add right and left to the list of potential breakdowns  ret
            end if
            rightwords = FindAllMultiwords(word.substring(I, word.length, n-1)
            if  rightwords.length  > 0 then
                combine left with each of the solutions in rightwords list and add to the list ret
            end if
        end if
    end for
end procedure
```

APPENDIX B

SURVEY QUESTIONNAIRE

Attempt #1: User's first login/ registration

[User will be presented with the consent form. Upon agreement, the user will be presented with an expanded "Create Account" field]

Create User Account:
Please create an account for use in this study. Use your FSU email address for your username. Assume you are creating an email account and you want your password to be strong enough. Try to create your password in a manner that you would normally do. You should take whatever steps you normally take to remember and protect your password. DO NOT provide passwords that you currently use for another service. All passwords will be saved and analyzed. DO NOT use this password elsewhere.

Email (Use your FSU email ending in "@my.fsu.edu")

Password (minimum of 8 characters)

Confirm Password:

[User will be presented with: "You have successfully created your account. Please answer the following questions"

Survey form:

1- What is your gender?

□ Female
□ Male
□ I prefer not to answer

2- What is your age?

□ younger than 18 years old
□ 18-24 years old
□ 25-34 years old
□ 35-44 years old
□ 45-54 years old
□ 55 years or older

3- Which of the following best describes your highest education level?

□ High School graduate
□ Some college, no degree
□ Associates degree
□ Bachelors degree
□ Graduate degree (Masters, Doctorate, etc.)
□ Other

4- What is your native language?

[User will be presented with: "You have successfully completed this part of the study. Please remember to come back soon for your second login!"]

Attempt #2: User's second login

[Upon successful login, user will be brought to this survey page]

Survey Form:

1- Are you majoring in or do you have a degree or job in computer science, computer engineering, information technology, or a related field?

□ Yes
□ No
□ I prefer not to answer

2- In what department are you majoring?

3- How many website username and passwords do you have, approximately?

□ Less than 5 accounts
□ 5-10 accounts
□ 11-20 accounts
□ More than 20 accounts

4- Do you try to create unique passwords for each different account?

□ Yes, I create a new password every time I create a new account or every time I have to change my password.
□ No, I use my old passwords that I have already created for my other accounts.
□ I mostly create a new password but sometimes use my old passwords also.

[User will be presented with: "You have successfully completed this part of the study. Please remember to come back soon for your third login!"]

Attempt #3: User third login, change password

[User will be prompted to login. This process is the same as in Attempt #2]
[Upon successful login, user will be prompted to change password as follows:]

Your password has expired and must be changed. Please choose a new password. Try to change your password in a manner that you would normally do. You should take whatever steps you normally take to remember and protect your password. DO NOT provide passwords that you currently use for another service. All passwords will be saved and analyzed. DO NOT use this password elsewhere.

Old Password:

New Password (minimum of 8 characters.)

Confirm New Password:

Survey Form:

1- How did you choose your new password? Were you influenced by any of the following? (Please check all that apply.)

□ Names of family members, relatives, close friends
□ Familiar numbers (birth date, telephone number, street address, employee number, etc.)
□ Songs, movies, television shows, books, poetry or games.
□ Scientific or other educational mnemonics
□ Sports teams and players
□ Names of famous people or characters
□ Words in a language other than English
□ Other (please specify):_____

2- When creating your new password, did you consider any of the following policies to make your password more secure? (Please check all that apply.)

□ Include numbers
□ Include upper case letters
□ Include symbols (such as "!" or "#")
□ Have 8 or more characters
□ Not contain dictionary words
□ Not containing a sequence of adjacent or repeated characters on your keyboard (e.g. "qwerty")
□ I did not consider any policy
□ Other (please specify)_____

3- Did you create your new password by slightly changing your old password for this website?

□ Yes
□ No

4- Is the password that you have just created one that you have used in the past?

□ Yes
□ No
□ Password has similarities to another password that I have used before.

5- If you created your new password based on one of your old passwords, which of the following changes did you consider? (Please check all that apply.)

Word part: □ Not applicable □ Changed completely □ Changed slightly □ Capitalized letters
Numbers: □ Not applicable □ Added digits □ Deleted digits □ Substituted digits
Special characters: □ Not applicable □ Added symbols □ Deleted symbols □Substituted symbols

Attempt #4: User's fourth login

[User will be prompted to login. This process is the same as in Attempt #2]
[Upon successful login, user will be brought to survey page]

Survey form:

1- How long have you been using a computer?

□ 0–2 years
□ 3–5 years
□ 6–10 years
□ More than 10 years

2- How do you usually create passwords for your accounts? (Please check all that apply.)

□ Randomly generate a password using special software or apps
□ Reuse a password that is used for another account
□ Modify a password that is used for another account
□ Create a new password using a familiar number or a name of a family member
□ Choose a word and substitute some letters with numbers of symbols (for example '@' for 'a')
□ Use a passphrase consisting of several words
□ Choose a phrase and use the first letters of each word
□ Other (please specify) _____

3- How do you store your passwords? Check all that apply.

□ I store my passwords in a regular file / document on my computer.
□ I store my passwords in an encrypted computer file.
□ I use password management software to securely store my passwords.
□ I store my passwords on my cellphone / smartphone.
□ I save my passwords in the browser.
□ I write down my password on a piece of paper.
□ No, I do not save my passwords. I remember them.

4- If you have any additional feedback about passwords or this survey, please enter your comments here.

[User will be presented with: "Thank you for your participation in our research study. You have now completed all the steps. You are entered into our drawing. You will need your password to check if you have won the prize. Further instructions will be provided via email. Best of luck!"]

APPENDIX C

HUMAN SUBJECT APPROVAL LETTERS

C.1 Approval Letter

Mon 3/10/2014 10:34 AM

To: Houshmand Yazdi, Shiva;

The Florida State University
Office of the Vice President For Research
Human Subjects Committee
Tallahassee, Florida 32306-2742

APPROVAL MEMORANDUM

Date: 3/10/2014

To: Shiva Houshmand Yazdi

Address: 4083
Dept.: COMPUTER SCIENCE
From: Thomas L. Jacobson, Chair
Re: Use of Human Subjects in Research
Novel extensions to probabilistic password cracking

The application that you submitted to this office in regard to the use of human subjects in the proposal referenced above have been reviewed by the Secretary, the Chair, and one member of the Human Subjects Committee. Your project is determined to be Expedited per per 45 CFR § 46.110(7) and has been approved by an expedited review process.

The Human Subjects Committee has not evaluated your proposal for scientific merit, except to weigh the risk to the human participants and the aspects of the proposal related to potential risk and benefit. This approval does not replace any departmental or other approvals, which may be required.

If you submitted a proposed consent form with your application, the approved stamped consent form is attached to this approval notice. Only the stamped version of the consent form may be used in recruiting research subjects.

If the project has not been completed by 3/9/2015 you must request a renewal of approval for continuation of the project. As a courtesy, a renewal notice will be sent to you prior to your expiration date; however, it is your responsibility as the Principal Investigator to timely request renewal of your approval from the Committee.

You are advised that any change in protocol for this project must be reviewed and approved by the Committee prior to implementation of the proposed change in the protocol. A protocol change/amendment form is required to be submitted for approval by the Committee. In addition, federal regulations require that the Principal Investigator promptly report, in writing any unanticipated problems or adverse events involving risks to research subjects or others.

By copy of this memorandum, the Chair of your department and/or your major professor is reminded that he/she is responsible for being informed concerning research projects involving human subjects in the department, and should review protocols as often as needed to insure that the project is being conducted in compliance with our institution and with DHHS regulations.

This institution has an Assurance on file with the Office for Human Research Protection. The Assurance Number is FWA00000168/IRB number IRB00000446.

Cc: Sudhir Aggarwal, Advisor
HSC No. 2014.12320

The formal PDF approval
letter: http://humansubjects.magnet.fsu.edu/pdf/printapprovalletter.aspx?app_id=12320

C.2 Re-Approval Letter

Mon 3/16/2015 4:16 PM

To: Houshmand Yazdi, Shiva;

The Florida State University
Office of the Vice President For Research
Human Subjects Committee
Tallahassee, Florida 32306-2742

RE-APPROVAL MEMORANDUM

Date: 3/16/2015

To: Shiva Houshmand Yazdi

Address: 4083
Dept.: COMPUTER SCIENCE
From: Thomas L. Jacobson, Chair
Re: Re-approval of Use of Human subjects in Research
Novel extensions to probabilistic password cracking

Your request to continue the research project listed above involving human subjects has been approved by the Human Subjects Committee. If your project has not been completed by 3/14/2016, you must request a renewal of approval for continuation of the project. As a courtesy, a renewal notice will be sent to you prior to your expiration date; however, it is your

responsibility as the Principal Investigator to timely request renewal of your approval from the committee.

If you submitted a proposed consent form with your renewal request, the approved stamped consent form is attached to this re-approval notice. Only the stamped version of the consent form may be used in recruiting of research subjects. You are reminded that any change in protocol for this project must be reviewed and approved by the Committee prior to implementation of the proposed change in the protocol. A protocol change/amendment form is required to be submitted for approval by the Committee. In addition, federal regulations require that the Principal Investigator promptly report in writing, any unanticipated problems or adverse events involving risks to research subjects or others.

By copy of this memorandum, the Chair of your department and/or your major professor are reminded of their responsibility for being informed concerning research projects involving human subjects in their department. They are advised to review the protocols as often as necessary to insure that the project is being conducted in compliance with our institution and with DHHS regulations.

Cc: Sudhir Aggarwal, Advisor
HSC No. 2015.15030

The formal PDF approval
letter: http://humansubjects.magnet.fsu.edu/pdf/printapprovalletter.aspx?app_id=15030

APPENDIX D

SAMPLE CONSENT FORM

My name is Shiva Houshmand, and I am a graduate student in the Department of Computer Sciences at the Florida State University. I am conducting a research study to understand how users create and manage their passwords. You have been invited to participate because you have confirmed that you are at least 18 years old.

If you agree to be in this study, you will be then asked to create a username and password to start with. You will be advised to NOT provide a password that you currently use or have previously used for another account. You will be asked a series of demographic questions such as age, education, and gender. You are required to log in once a day for a total of four times during the period of the study. Each time you log in, some multiple-choice questions will be asked related to how you create and manage your passwords. You may be asked to change your password during the logins. Each login should not take more than 5 minutes. If you finish all four days of this survey, you will be entered into a drawing in which you can win a $25 worth Amazon gift card. The drawing will be held at the end of the study and the winners will be notified via email.

All questionnaire responses and passwords will only be retained for the duration of the study. Only researchers in this study will have access to the data. To maintain the confidentiality of your records, the password or your answers will not be associated with your email address or any other identifiable information. Your email address is only used to send follow up emails for the result of the drawing and will be discarded after the duration of the study. The results of this research study may be published, but only aggregate data will be reported. The records of this study will be kept private and confidential to the extent permitted by law.

There are no known risks if you decide to participate in this research study. Your participation in this study is completely voluntary. Your decision whether or not to participate will not affect your current or future relations with the University. If you choose to participate, you are free to withdraw at any time without penalty or risk. Participating in this research study may not benefit you directly, but what we learn from this study would provide general benefits to the password security area.

The researchers conducting this study are Shiva Houshmand and her advisor Dr. Sudhir Aggarwal. If you have any questions or concerns about the questionnaire or about being in this study, you may contact them at (---) ------- or at -----@my.fsu.edu, and ------@cs.fsu.edu. If you have any questions or concerns regarding this study and would like to talk to someone other than the researchers, you are encouraged to contact the FSU IRB at ---- ---- -----, -------- --------, ----- ---, Tallahassee, FL, or (---) -------, or by email at ----------@magnet.fsu.edu.

By clicking I Accept, you confirm that you have read the above information, you have asked any questions you may have had and have received answers, and you consent to participate in the study.

APPENDIX E

PSEUDO-CODE FOR TARGETED ATTACK

E.1 Algorithm for Modeling Differences

```
// find edit distance of two password strings using hierarchical algorithm, then determine if any
//of the most common changes have been applied and create a targeted grammar. Finally merge
//the targeted grammar with a comprehensive grammar.

procedure MODELDIFFERENCES
    simple_base1= PARSESIMPLEBASESTRUCTURE (pass1)
    simple_base2= PARSESIMPLEBASESTRUCTURE (pass2)
    simple_transform = DL_EDITDISTANCE ( simple_base1, simple_base2 )
    level1_distance = simple_transform.distance
    if  simple_transform.contains("t")  then
        changed_pass1 = REVERTTRANSPOSITION()
    else
        changed_pass1 = pass1
    end if
    transform = DL_EDITDISTANCE ( changed_pass1, pass2 )
    level2_distance = transform.distance

    // Create Target Grammar based on changes found
    if  base structures are the same  then
        add base structure to TGrammar
    end if
    if  L components are the same  then
        add both capitalizations to TGrammar
    end if
    if  transform.contains("s")  and component is digit then
        if  number is incremented   then
            add (number + 1) to TGrammar
        else if  number is decremented  then
            add (number – 1) to TGrammar
        end if
    end if
    if  transform.contains("i")  and  component is digit  and  number is being repeated  then
        add the repeated number and the new base structure to TGrammar
    end if
    MERGEGRAMMAR (TGrammar, InitialGrammar)
end procedure
```

E.2 Computation of Damerau-Levenshtein Edit Distance

```
procedure DL_EDITDISTANCE (pass1, pass2)
    for (i = 0 ; i <= pass2.length + 1 ; i++) do              //fill the first column
        dist[i][0] = i
        operation[i][0] = 'i'
    end for
    for (j = 0 ; j <= pass1.length + 1 ; j++) do              //fill the first row
        dist[0][j] = j
        operation[0][j] = 'd'
    end for
    for ( i = 0 ; i < pass2.length ; i++ ) do
        for ( j = 0 ; j < pass1.length ; j++ ) do
            cost = (pass2[i] == pass1[j]) ? 0 : 1
            if   dist[i + 1][j] + 1  ≤   dist[i][j + 1] + 1  then
                if  dist[i + 1][j] + 1  ≤  dist[i][j] + cost  then
                    dist[i + 1][j + 1] =  dist[i + 1][j] + 1
                    operation[i + 1][j + 1]  =  'd'                  //deletion
                else
                    dist[i + 1][j + 1] = dist[i][j] + cost
                    operation[i + 1][j + 1] = 's'                    //substitution
                end if
            else
                if  dist[i][j + 1] + 1  ≤ dist[i][j] + cost  then
                    dist[i + 1][j + 1]  =  dist[i][j + 1] + 1
                    operation[i + 1][j + 1] = 'i'                    //insertion
                else
                    dist[i + 1][j + 1] = dist[i][j] + cost
                    operation[i + 1][j + 1] = 's'                    //substitution
                end if
            end if
            if   pass2[i] == pass1[ j – 1]   and   pass2[i – 1] == pass1[j]   then
                operation[i + 1][j + 1] = 't'                        //transposition
                if  dist[i + 1][j + 1]  ≤  dist[i - 1][j - 1] + cost  then
                    dist[i + 1][j + 1] = dist[i + 1][j + 1]
                else
                    dist[i + 1][j + 1] = dist[i - 1][j - 1] + cost
            end if
        end for
    end for
    result = FINDOPERATIONS()
    return  result
end procedure
```

E.3 Computation of Damerau-Levenshtein Backtracking

```
procedure FINDOPERATIONS ()
    n = pass2.length;
    m = pass1.length;

    while  ( n > 0  ||  m > 0 )  do
        if  ( operation[n][m] ==  'i'  ) then
            op = "i" + op
            n = n -1                                        //UP
        else if  ( operation[n][m] ==  'd' ) then
            op = "d" + op
            m = m -1                                        //LEFT
        else if  ( operation[n][m] == 's' ) then
            if  ( dist[n - 1][m - 1] == dist[n][m] ) then
                op = "n" + op;
            else
                op = "s" + op;
            end if
            n = n -1                                        //DIAG
            m = m -1
        else if  ( operation[n][m] == 't' ) then
            op = "t" + op;
            n = n - 2                                       //DIAG TWO CELLS
            m = m - 2
        end if
    end while
    return op
end procedure
```

REFERENCES

[1] Matt Weir, Sudhir Aggarwal, Breno de Medeiros, Bill Glodek, "Password Cracking Using Probabilistic Context Free Grammars," Proceedings of the 30th IEEE Symposium on Security and Privacy, May 2009.

[2] W. Burr, D. Dodson, R. Perlner, W. Polk, S. Gupta, E. Nabbus, "NIST special publication 800-63-1 electronic authentication guideline," National Institute of Standards and Technology, Gaithersburg, MD, April, 2006.

[3] Weir, Matt, Sudhir Aggarwal, Michael Collins, and Henry Stern. "Testing metrics for password creation policies by attacking large sets of revealed passwords." In Proceedings of the 17th ACM conference on Computer and communications security, pp. 162-175. ACM, 2010.

[4] E. R. Verheul, "Selecting secure passwords," CT-RSA 2007, LNCS 4377, pp 49-66, 2007.

[5] Shiva Houshmand, "Analyzing Password Strength and Efficient Password Cracking", Electronic Thesis, Treatises and Dissertations. Paper 3737, June 2011.

[6] D. Goodin. Why Passwords have never been weaker and crackers have never been stronger, 2012. [Online]. Ars Technica. Available: http://arstechnica.com/security/2012/08/passwords-under-assault/, accessed June 8, 2015.

[7] M. Weir, Using Probabilistic Techniques to aid in Password Cracking Attacks, Dissertation, Florida State University, 2010.

[8] RainbowCrack [Online]. available: http://project-rainbowcrack.com/table.htm, accessed June 8, 2015.

[9] AirCrack [Online]. Available: http://www.aircrack-ng.org/, accessed June 8, 2015.

[10] Oechslin ,P. 2003. Making a faster cryptanalytic time-memory trade-off. Advances in Cryptology—CRYPTO 2003. Lecture Notes in Computer Science, vol. 2729. Springer, 617–630

[11] PasswordLastic [Online]. Available: http://www.passwordlastic.com, accessed June 8, 2015.

[12] THC Hydra [Online]. Available: http://www.thc.org/thc-hydra/, accessed June 8, 2015.

[13] Ncrack [Online]. Available: http://nmap.org/ncrack/, accessed June 8, 2015.

[14] Medusa [Online]. Available: http://www.foofus.net/~jmk/medusa/medusa.html, accessed June 8, 2015.

[15] Fgdump [Online]. Available: http://www.foofus.net/~fizzgig/fgdump/, accessed June 8, 2015.

[16] Brutus [Online]. Available: http://www.hoobie.net/brutus/, accessed June 8, 2015.

[17] L0phtcrack [Online]. Available: http://www.l0phtcrack.com/, accessed June 8, 2015.

[18] Cain and Able [Online]. Available: http://www.oxid.it/cain.html, accessed June 8, 2015.

[19] ElcomSoft [Online]. Available: http://www.elcomsoft.com/, accessed June 8, 2015.

[20] AccessData [Online]. Available: http://www.accessdata.com/, accessed June 8, 2015.

[21] John the Ripper [Online]. Available: http://www.openwall.com/john/, accessed June 8, 2015.

[22] Hashcat [Online]. Available: http://hashcat.net/oclhashcat/, accessed June 8, 2015.

[23] T. Booth and R. Thompson, "Applying Probability Measures to Abstract Languages," *IEEE Transactions on Computers*, Vol. C-22, No. 5, May 1973.

[24] A. Narayanan and V. Shmatikov, "Fast Dictionary Attacks on Passwords Using Time-Space Tradeoff," in Proceedings of the 12th ACM conference on Computer and communications security, ser. CCS '05, 2005, pp. 364–372.

[25] J. Ma, W. Yang, M. Luo, and N. Li, "A study of probabilistic password models," in *Proc. 35th IEEE Symp. Secur. Privacy*, May 2014, pp. 689–704.

[26] P. G. Kelley, S. Komanduri, M. L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, and J. Lopez, "Guess Again (and Again and Again): Measuring Password Strength by Simulating PasswordCracking Algorithms," in Proceedings of the 33rd IEEE Symposium on Security and Privacy, ser. SP '12, 2012, pp. 523–537.

[27] C. Castelluccia, M. Durmuth, D. Perito, "Adaptive password-strength meters from Markov models," NDSS '12, 2012.

[28] R. Veras, C. Collins, and J. Thorpe, "On the semantic patterns of passwords and their security impact," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, Feb. 2014.

[29] Yahoo Credentials. [Online]. Available: http://news.cnet. com/8301-1009_3-57470786-83/hackers-post-450k-credentials-pilfered- from-yahoo, accessed June 8, 2015.

[30] A. Vance, "If your password is 123456, just make it hackme," [Online]. New York Times, January 2010, Available: http://www.nytimes.com/2010/01/21/technology/21password .html, accessed June 8, 2015.

[31] Robert McMillan, "Phishing attack targets MySpace users," [Online]. Available: http://www.infoworld.com/d/security-central/phishing-attack-targets-myspace-users-614, October 27, 2006, accessed June 8, 2015.

[32] T. Warren, “Thousands of Hotmail Passwords Leaked,” http://www.neowin.net/news/main/09/10/05/thousands-of-hotmail-passwords-leaked-online.

[33] Six Million Users’ Privacy Leaked. [Online]. Available: http://www.china- online-marketing.com/news/anti-virus-news/csdn-tianya-renren-kaixin- hacked-6-million-users-privacy-leaked/, accessed June 8, 2015.

[34] A list of popular password cracking wordlists, [Online]. Available: http://www.outpost9.com/files/WordLists.html, accessed June 8, 2015.

[35] EOWL [Online]. Available: http://dreamsteep.com/projects/the-english-open-word-list.html, accessed June 8, 2015.

[36] Most common male and female first names in the US [Online]. Available: http://names.mongabay.com/, accessed June 8, 2015.

[37] Top 40,000 words from TV and movie scripts [Online]. Available: http://en.wiktionary.org/wiki/Wiktionary:Frequency_lists#TV_and_movie_scripts, accessed June 8, 2015.

[38] Shiva Houshmand, Sudhir Aggarwal, and Randy Flood. “Next Gen PCFG Password Cracking.” *IEEE Transactions on Information Forensics and Security*, April 2015.

[39] De Luca, Alexander, Roman Weiss, and Heinrich Hussmann. "PassShape: stroke based shape passwords." In Proceedings of the 19th Australasian Conference on Computer-Human interaction: Entertaining User interfaces, pp. 239-240. ACM, 2007.

[40] Schweitzer, Dino, Jeff Boleng, Colin Hughes, and Louis Murphy. "Visualizing keyboard pattern passwords." In Visualization for Cyber Security, 2009. VizSec 2009. 6th International Workshop on, pp. 69-73. IEEE, 2009.

[41] Detlef Prescher, “A Tutorial on the Expectation-Maximization Algorithm Including Maximum-Likelihood Estimation and EM Training of Probabilistic Context-Free Grammars,” *CERN Document Server*, preprint, 2004.

[42] C. Manning and H. Schuetze, Foundations of Statistical Natural Language Processing. MIT Press, 1999.

[43] Bonneau, Joseph. "The science of guessing: analyzing an anonymized corpus of 70 million passwords." In Security and Privacy (SP), 2012 IEEE Symposium on, pp. 538-552. IEEE, 2012.

[44] Klein, Daniel V. "Foiling the cracker: A survey of, and improvements to, password security." In Proceedings of the 2nd USENIX Security Workshop, pp. 5-14. 1990.

[45] M. Dell'Amico, P. Michiardi, and Y. Roudier, "Password Strength: An Empirical Analysis," in Proceedings of the 29th conference on 13 Information Communications, ser. INFOCOM'10, 2010, pp. 983–991.

[46] Michael Garey and David S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," W.H Freeman and Co. 1979.

[47] R. Waugh, No wonder hackers have it easy: Most of us now have 26 different online accounts - but only five passwords. [Online]. Available: http://www.dailymail.co.uk/sciencetech/article-2174274/No-wonderhackers-easy-Most-26-different-online-accounts-passwords.html, accessed June 8, 2015.

[48] D. Florencio and C. Herley, "A Large-Scale Study of Web Password Habits," in Proceedings of the 16th International Conference on the World Wide Web, 2007, pp. 657–666.

[49] R. Shay, S. Komanduri, P. G. Kelley, P. G. Leon, M. L. Mazurek, L. Bauer, N. Christin, and L. F. Cranor. Encountering stronger password requirements: User attitudes and behaviors. In 6th Symposium on Usable Privacy and Security, July 2010.

[50] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang, "The Tangled Web of Password Reuse," NDSS'14, February 23-26.

[51] Y. Zhang, F. Monrose, M. Reiter, "The security of modern password expiration: an algorithmic framework and empirical analysis," Proceeding of CCS'10, October 4-8, 2010, pp. 176-186.J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[52] F. J. Damerau, "A Technique for Computer Detection and Correction of Spelling Errors," Commun. ACM, vol. 7, no. 3, pp. 171–176, Mar. 1964.

[53] Shiva Houshmand, and Sudhir Aggarwal. "Building Better Passwords using Probabilistic Techniques." *ACSAC'12: Proceedings of the 28th Annual Computer Security Applications Conference*, December 2012, pages 109-118.

[54] Shannon Riley. Password Security: What Users Know and What They Actually Do. Usability News, 8(1), 2006.

[55] Furnell, S., "An assessment of website password practices," Computers & Security 2007 445-451.

[56] X. de Carné de Carnavalet and M. Mannan, "From very weak to very strong: Analyzing password-strength meters." NDSS '14, 23-26 February 2014,

[57] Philip G. Inglesant, M. Angela Sasse, "The true cost of unusable password policies: password use in the wild," Proc. of the 28th international conference on Human factors in computing systems, April 10-15, 2010, Atlanta, Georgia.

[58] Charoen, D., Raman, M., and Olfman, L., “Improving end user behavior in password utilization,” Systemic Practice and Action Research, 21(1), 55. 2008.

[59] A. Adams and M. A. Sasse, “Users are not the enemy,” Communications of the. ACM, 42(12):40–46, 1999.

[60] S. Schechter, C. Herley, M. Mitzenmacher, “Popularity is everything: a new approach to protecting passwords from statistical-guessing attacks”, HotSec'10: Proceedings of the 5th USENIX conference on Hot Topics in Security, Aug 2010.

[61] Kaspersky Labs, Perception and Knowledge of IT threats: the consumer’s point of view. [Online]. Available: http://www.kaspersky.com/downloads/pdf/kaspersky-lab_ok-consumer-survey-report_eng_final.pdf, accessed June 8, 2015.

[62] Shiva Houshmand, Sudhir Aggarwal, and Umit Karabiyik. “Identifying Passwords Stored on Disk.” *Eleventh IFIP WG 11.9 International Conference on Digital Forensics*, January 2015.

[63] Detective Steve Grimm, Webster Grove Police Department, personal communication 2014.

[64] C. Hargreaves, H. Chivers, Recovery of encryption keys from memory using a linear scan, Third International Conference on Availability, Reliability and Security, pp. 1369-1376, 2008.

[65] S. Lee, A. Savoldi, S. Lee and J. Lim, Password recovery using an evidence collection tool and countermeasures, In proceedings of Intelligent Information Hiding and Multimedia Signal Processing, pp. 97-102, 2007.

[66] D. Riis, Google Chrome Password Recovery. [Online]. Available: http://bitbucket.org/Driis/chromepasswordrecovery, accessed June 8, 2015.

[67] S. Garfinkel, N. Beebe, L. Liu, and M. Maasberg, Detecting thretenting insiders with lightweight media forensics, Technologies for Homeland Security (HST), pp. 86-92, 2013.

[68] Guidance Software, Encase. [Online]. Available: http://www.guidancesoftware.com/, accessed June 8, 2015.

[69] Identity Finder, Sensitive Data Manager. [Online]. Available: http://www.identityfinder.com/us/Business/IdentityFinder/SensitiveDataManager, accessed June 8, 2015.

[70] The Sleuth Kit. [Online]. Available: http://www.sleuthkit.org, accessed June 8, 2015.

[71] Apache OpenNLP Sentence detection tool. [Online]. Available: https://opennlp.apache.org, accessed June 8, 2015.

[72] Digital Corpora Govdocs1 [Online]. Available:http://digitalcorpora.org, accessed June 8, 2015.

BIOGRAPHICAL SKETCH

Shiva Houshmand is a Ph.D. candidate in Computer Science at Florida State University. She received her B.Sc. in 2008 from the University of Tehran and her M.S. in 2011 from Florida State University both in Computer Science. Her research interests include computer and network security, authentication, usable security, digital forensics and machine learning. She has been working on developing metrics for analyzing password strength and probabilistic techniques for more effective password cracking. She is also interested in security and privacy issues in the Internet and has been working on search, entity resolution and identification problems in the Internet.