Modeling the adversary to evaluate password strength with limited samples

Saranga Komanduri

CMU-ISR-16-101
February 2016

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Lorrie Faith Cranor, Chair
Lujo Bauer
Nicolas Christin
Paul C. van Oorschot, Carleton University

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation under grant nos. DGE-0903659 and CNS-1116776, and by generous donations from Microsoft Research.

The views and conclusions contained in this document are those of the author, and should not be interpreted as representing the official policies, either expressed or implied, of Microsoft, the U.S. government, or any other entity.

Keywords: passwords, security, usability, policy, modeling, statistics

Abstract

In an effort to improve security by preventing users from picking weak passwords, system administrators set password-composition policies, sets of requirements that passwords must meet. Guidelines for such policies have been published by various groups, including the National Institute of Standards and Technology (NIST) in the United States, but this guidance has not been empirically verified. In fact, our research group and others have discovered it to be inaccurate.

In this thesis, we provide an improved metric for evaluating the security of password-composition policies, compared to previous machine-learning approaches. We make several major contributions to passwords research. First, we develop a guess-calculator framework that automatically learns a model of adversary guessing from a training set of prior data mixed with samples, and applies this model to a set of test passwords. Second, we find several enhancements to the underlying grammar that increase the power of the learning algorithm and improve guessing efficiency over previous approaches. Third, we use the guess-calculator framework to study the guessability of passwords under various policies and provide methodological and statistical guidance for conducting these studies and analyzing the results. While much of this thesis focuses on an offline-attack threat model in which an adversary can make trillions of guesses, we also provide guidance on evaluating policies under an online-attack model, where the user can only make a small number of guesses before being locked out by the authentication system.

Dedication

March 20, 2015. That was the day that changed my life forever. That was the day my wife and I welcomed our son, Shreyas Jishnu, to the world. This thesis is dedicated to Shreyas. As I think about all the things I want for Shreyas, I have a new lens through which I see the world and give thanks . . .

I hope to show Shreyas love and support through all his good days and his bad days. For this, I need to thank my own parents. They tolerated my mischief, supported me through difficult days as a teenager and encouraged me to aim high and achieve whatever I put my mind to. Speaking of parents, I need to thank my mother- and father-in-law for treating me like their own son ever since the day my wife and I decided to get married, and for stepping in to help when life became too crazy for us to manage alone.

I hope Shreyas can find caring, supportive, and smart friends to grow up with. I now spend great time and effort trying to understand how best to raise him so he learns the right lessons and ultimately makes good decisions for himself. In my own life, I need to thank my long-time friends for always having my back when I needed them the most: Casey, Brady, Rob, and Jason. It's great to now see all of our families grow together and I look forward to many more fun times ahead. In addition, I thank the many friends who made life at CMU so much more enjoyable: Rich, Cristian, Kami, Michelle, Rebecca, Pedro, Blase, and Manya.

I hope Shreyas will find great mentors to counsel and help him navigate his future. Lorrie Cranor did this for me and, as I recount my long journey as a graduate student, I have much to thank her for. Lorrie is not only a smart and talented scientist, she is also a teacher, guide, and good friend. She watches out for each of her students as though we are her own children. This included helping us learn and think like scientists, allowing us to present our successes at conferences, providing feedback on areas we could improve, and even hosting us in her home for holiday parties. For the first time, I realize how much work that must be. Lorrie always wanted what was best for me, both professionally and personally, and I sincerely thank her for her thoughtfulness, mentorship, sensitivity and time. I also would like to thank Lujo Bauer and Nicolas Christin for their time, guidance, and confidence in my work throughout the years, and Paul van Oorschot for

the time he invested as one of my thesis committee members. Paul's suggestions improved this thesis in ways I never expected. I also need to thank Jessica Staddon from Google and Stuart Schechter from Microsoft Research for wonderful collaboration and internship opportunities that set me up for future success.

I hope Shreyas will some day find work that is fulfilling to him and join an institution that supports him both professionally and personally. I have to thank Gabriel Burt and Jenny Farver for giving me an opportunity to join the Civis Analytics team. More importantly, I have to thank them for making me feel like my contributions are valuable. I have to thank them for looking out for my personal well-being, which included allowing me the time I needed to work on my thesis, covering my time off when Shreyas was born, and showing tremendous patience as daycare germs in our household took me out of commission over and over.

I hope that Shreyas will find love. The first time I found this was with my wife, Pradipta, who always supports me and does everything possible to take care of our family. I never imagined that I would some day have true love twice—the second time in Shreyas, who enriches our lives in too many ways to describe.

I pray that Shreyas will have hope, optimism, curiosity and faith as he begins his journey in this world. Lastly, but most importantly, I have to thank God for my family and all the blessings we have. Completing this thesis is a milestone that allows us to move on to new adventures together—I am excited to see what the future has in store for us.

Saranga Komanduri
October 21, 2015

Table of Contents

Chapter 1

Introduction

There are three actors that we can consider in the password-authentication ecosystem. There are users who create passwords, adversaries who want to impersonate users, and system administrators who try to block adversaries and protect users' passwords. Among the tools that system administrators have at their disposal are password policies. Password policies are sets of constraints on passwords, e.g., that they must contain a number or will expire every 90 days.

A subset of password policies are *password-composition policies*,[1] which are requirements on the literal passwords created by users. For example, a password-composition policy might require some number of symbols or force the password to pass a dictionary check. The purpose of password-composition policies is, presumably, to make passwords harder to guess, but we do not have a good understanding of how effective these policies are at protecting users. This is the motivation behind my thesis: to improve upon existing metrics for evaluating the strength of password-composition policies.

These policies are important because they impose costs on users. We have found that when password-composition policies get more complex, users get more frustrated and have a harder time remembering their passwords [128]. We have also found evidence that security and usability are not always optimized. Our research, based partly on methods developed in this thesis, suggests that policies that mandate long passwords are more secure **and** more usable than policies that mandate shorter passwords with more complexity requirements [128]. These apparently suboptimal decisions are not the fault of system administrators—guidance in this area is simply lacking. In 2006, the National Institute of Standards and Technology (NIST) published guidelines for setting password policies [21]. This guidance was based on many assumptions about user behavior that our

[1] A note on presentation: we identify terms that we define in this thesis in italics, and terms from other sources with quotation marks. This convention is followed throughout the thesis.

research group subsequently found to be inadequate [129] which led to inaccurate estimates of password strength [67,76]. The guidelines were revised in 2011 to remove many of the earlier recommendations [19], and new guidance was not inserted in its place. The initial NIST guidelines had broad impact, however, and were cited as an authority in choosing the password policies of several organizations, including Carnegie Mellon University [95,129]. Based on NIST's guidance, CMU mandated shorter, 8-character passwords with many complexity requirements instead of longer passwords with fewer requirements.

Passwords compared with other schemes

System administrators are justified in trying to improve the security of their users' passwords. Like many information-security schemes, password authentication relies on a secret for its security. Possession of a password is often all that is needed to authenticate a user.[2]

As a point of comparison, consider SSL certificates. They also rely on a secret for security. When you interact with a website over SSL, your browser verifies that the domain name you visit matches the domain name on the website's SSL certificate. The certificate is trusted if it is signed by a certificate authority (CA).[3] This signature is created using a "private key." Were another party to obtain this key, they could masquerade as the certificate authority. Possession of the secret private key is enough proof to convince you, and any other users, of the website's identity.

A crucial difference between the CA's private key and a password is in the *guessability* of these keys. The CA's private key is an algorithmically generated, large number that is extremely hard to guess, i.e., the probability that an attacker with no special knowledge of the key will guess it is nearly zero, even with trillions of guesses. Passwords are chosen by humans who must expend a great deal of effort to remember random strings [10]. As we will show, trillions of guesses are sometimes enough to guess more than 89% of passwords. This is especially worrisome given the number of high-profile password breaches in recent years [32,68,70,71,82,103,112,122,131,141]. Even though the passwords in these breaches were hashed, they are still vulnerable to a "guessing attack" in which the adversary generates guesses and can verify these guesses using the password file [99].

[2] Authentication can be defined as "the act of establishing confidence in the identities of users" [20]. Claiming a specific identity usually begins with a non-secret token, such as a username or email address.

[3] For a detailed description of what is commonly referred to as SSL, see `http://security.stackexchange.com/a/20847`.

The insecurity of passwords is well known, at least in the security community, but passwords are still a popular form of authentication. One might wonder whether passwords are still relevant, or will be replaced by other technologies soon. A wide array of alternative forms of authentication have been proposed, such as biometrics [2], graphical password schemes [4], multi-factor authentication [108], schemes that use a smartphone [91], keystroke dynamics [104], and so on, but simple text passwords are still used extensively on websites and in the enterprise. Given the various hurdles that must be overcome for alternative authentication systems to be adopted, some researchers argue that the need for users to create and use text passwords will not be going away any time soon [8, 58]. We agree with this claim, but hope that this thesis will emphasize the need for alternative authentication systems that are not solely reliant on human memory.

Section 1.1 presents our threat model. In Section 1.2, we explain what makes studying passwords challenging. In Section 1.3, we present our thesis statement, which describes our high-level goals and approach. In Section 1.4, we list the contributions of this thesis. In Section 1.5, we provide a list of previous papers where these contributions have appeared before and describe how this thesis relates to that material. In Section 1.6, we provide a detailed outline of this thesis. Finally, in Section 1.7, we provide links to the software developed in this thesis along with guidance on reproducing our experiments.

1.1 Threat model

Before we can discuss protecting users, we need to specify what we are protecting them from. In this section, we present our threat model, which is a set of assumptions about our adversary. First we introduce some terminology, then we list and discuss the main assumptions of our model.

When adversaries try to guess a set of passwords, we say that they are making guesses *against a target policy*, where the target policy is the password-probability distribution of the target.[4] When a password is guessed successfully, we say that password has been *cracked*. *Guessing efficiency* refers to the rate of success of a guessing attack per guess, i.e., the average proportion of passwords cracked per guess.

[4]Password-probability distributions are described in Section 1.2.

Assumption: Passwords must be cracked

Our threat model assumes that the adversary can only identify users' passwords by guessing potential passwords and then invoking some process to determine if a guess matches a user's password. The typical example of this is cracking a file of hashed passwords. To identify passwords, the adversary must generate guesses, hash them, and compare the resulting hashes to the hashed passwords in the file. This is also known as a "guessing attack" [99].

We believe this assumption is realistic. Many websites have recently had hashed password files stolen [32, 68, 70, 71, 82, 103, 112, 122, 131, 141], which suggests that adversaries can sometimes obtain these files.

That said, there are plenty of scenarios in which passwords do not need to be cracked. Bonneau and Preibusch found that many authentication systems store or transmit passwords in cleartext [9]. On the user side, there are many classes of attacks that can obtain passwords in cleartext such as shoulder-surfing, malware, and social engineering attacks [4]. In all of the above cases, the strength of passwords is irrelevant. The adversary has no need to guess the password if it can be retrieved in cleartext from the user, authentication client, server, or in transit. These scenarios are clearly important and should be addressed by system administrators and system designers, but they are not covered by our threat model.

Assumption: The adversary can make $> 10^{12}$ guesses

We assume that the adversary can make over one trillion guesses without being caught or blocked.

Professional password crackers use tools that can make at least 350 billion guesses per second for certain hashing algorithms [54], and there is nothing preventing an adversary from having an equivalent level of computational power.

Assumption: Limited knowledge

The adversary's knowledge is limited to: the password-composition policy of the target; datasets of passwords; and other relevant data, such as dictionaries. We assume this data is not orders of magnitude larger than datasets we know about.

This assumption has a few implications. First, the adversary has no special knowledge of individual users or their passwords, and so cannot launch targeted attacks on individual passwords. Second, since we assume that the adversary does not have a dataset with trillions of passwords with which to make guesses, we expect adversaries to adopt some strategy for generating new passwords.

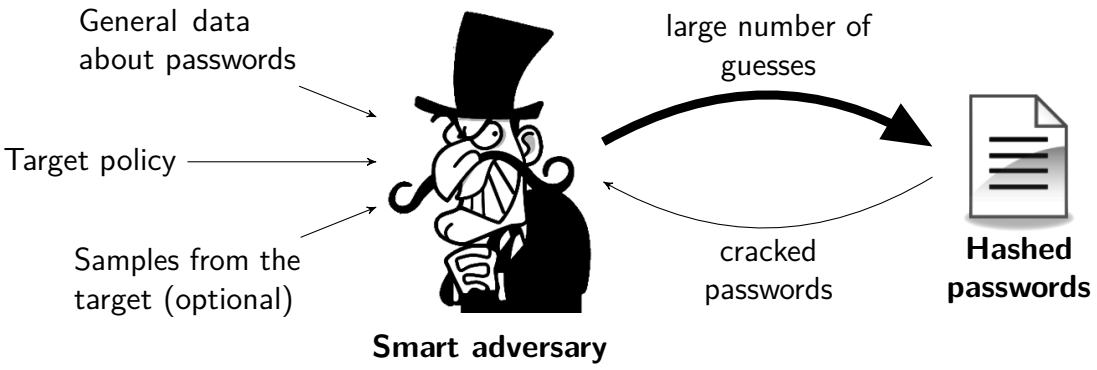


Figure 1.1: Overview of the threat model we assume in this thesis. The adversary makes a large number of guesses based on data and knowledge of the target password-composition policy. The *smart* adversary cares about guessing efficiency and will try to optimize their guessing strategy to crack passwords as quickly as possible. Guesses are made against a file of hashed passwords.

We believe this is a more realistic model than in some previous work such as Bonneau [7], discussed in Section 2.5.2.

Assumption: Guessing efficiency

Finally, we assume that the adversary cares about efficiency and will try to optimize their guessing strategy to crack passwords as quickly as possible.

Since each unsuccessful guess wastes computational effort and this effort has some cost to the adversary, if only an opportunity cost, we assume that the adversary is concerned with efficiency.

Figure 1.1 provides a visual summary of our threat model: our adversary makes a large number of guesses against hashed passwords, based on limited knowledge, and is concerned with the efficiency of these guesses.

The concepts developed in this thesis could be applied in other scenarios, such as an "online attack" where the adversary can only make a small number of guesses before being locked out, but we did not specifically target such a scenario in this thesis.

Plaintext passwords Our threat model allows the adversary to learn from datasets of plaintext passwords, but they are forced to make guesses against hashed passwords. Obviously, if the adversary had plaintext passwords, they would not need to make guesses! Therefore, it might be surprising that the approach we develop in this thesis requires plaintext passwords to be evaluated. In

other words, our approach requires that we (researchers) use a test set of *plaintext* passwords to model how the adversary would perform if they were cracking *hashed* passwords.

Having plaintext passwords allows our approach to model many more guesses in the same amount of computation time than would be required if we needed to crack hashed passwords. This aspect of our implementation allows us to emulate adversaries with far more computational power than we have, using techniques described in Chapter 4.

1.2 Probability distributions of passwords

Currently, system administrators do not have proper guidance in choosing among password-composition policies. The goal of this thesis is to improve upon previous work in this area, so that administrators can make more informed choices. One might ask, however, why are existing approaches insufficient to the task? Why are complex approaches required at all? The answer lies in the probability distributions of passwords. These distributions are *sparse*, meaning that most of the probabilities are near zero. This makes passwords difficult to study empirically.

Under our threat model of a guessing attack, where the adversary only has general data about the passwords of the target, the most accurate way to characterize password strength for a given policy is by looking at the probability distribution of passwords. The most secure probability distribution has a uniform shape, where all passwords are equally likely. Figure 1.2 plots a probability distribution from the RockYou dataset of leaked passwords, and shows that real password data is far from uniform. The most common passwords in this dataset, which happen to be "123456" and "password," are orders of magnitude more likely than almost all other passwords.

This illustrates why user-chosen passwords are much less secure than algorithmically generated secrets. If an adversary guessed "123456" and then "password" if the first guess failed, they could crack over 1% of passwords! To fix this, one might try banning common passwords. However, it is not clear that the resulting distribution would be more secure than the original. Theoretically, users who previously chose "123456" and "password" might choose "1234567" and "password1" respectively, creating a distribution that is less uniform than before! We need to measure the new distribution to evaluate its strength.

Once we have measured a distribution, the evaluation is theoretically straightforward. We can derive a simple metric parameterized by the number of guesses that the adversary will make, provided we are given a distribution that is

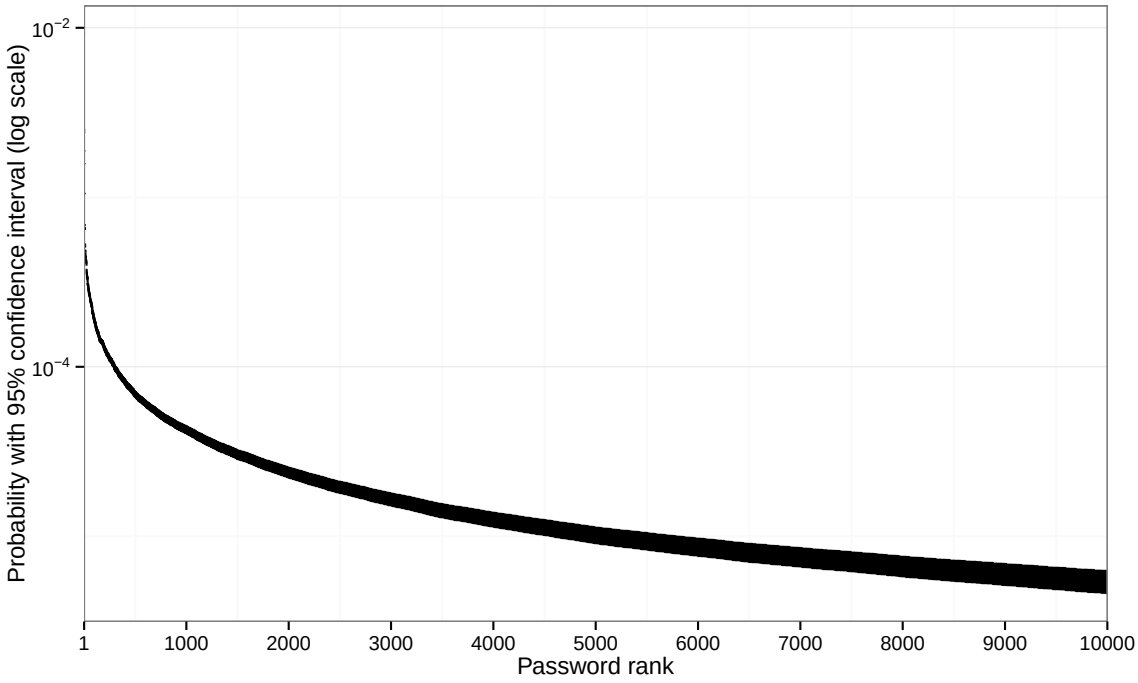


Figure 1.2: The top 10,000 passwords in the RockYou dataset ranked by probability. For each password, its observed probability with 95% confidence interval is plotted as a single vertical line. The combination of these produces a thick curve, widening to the right. As password frequency decreases, confidence in its true probability also decreases, resulting in larger confidence intervals. We cleaned this dataset as described in Appendix B.4.3.

ordered by decreasing probability. This metric is defined by way of example: given two policies with two probability distributions, $\mathcal{A}$ and $\mathcal{B}$, if the adversary makes ten guesses, we can sum the top ten probabilities in $\mathcal{A}$ and the top ten probabilities in $\mathcal{B}$ and compare the sums. The distribution with the lower sum is more secure against a guessing attack, because a smaller percentage of users' passwords are cracked. Given two password-probability distributions and a number of guesses, we can use this metric to decide which distribution is more secure.

However, this approach falls apart in practice when we increase the number of guesses we expect the adversary to make and draw confidence intervals around the probabilities that we have measured. Figure 1.3 shows confidence intervals around the top twenty passwords from Figure 1.2. Suppose we increase the number of guesses to one million. The RockYou dataset has over 14 million distinct passwords, yet our adversary would only have 95% confidence in the first seven guesses! As shown in Figure 1.3, the eighth and ninth guesses have overlapping confidence intervals, so we are unsure which password has a higher true probability. In other

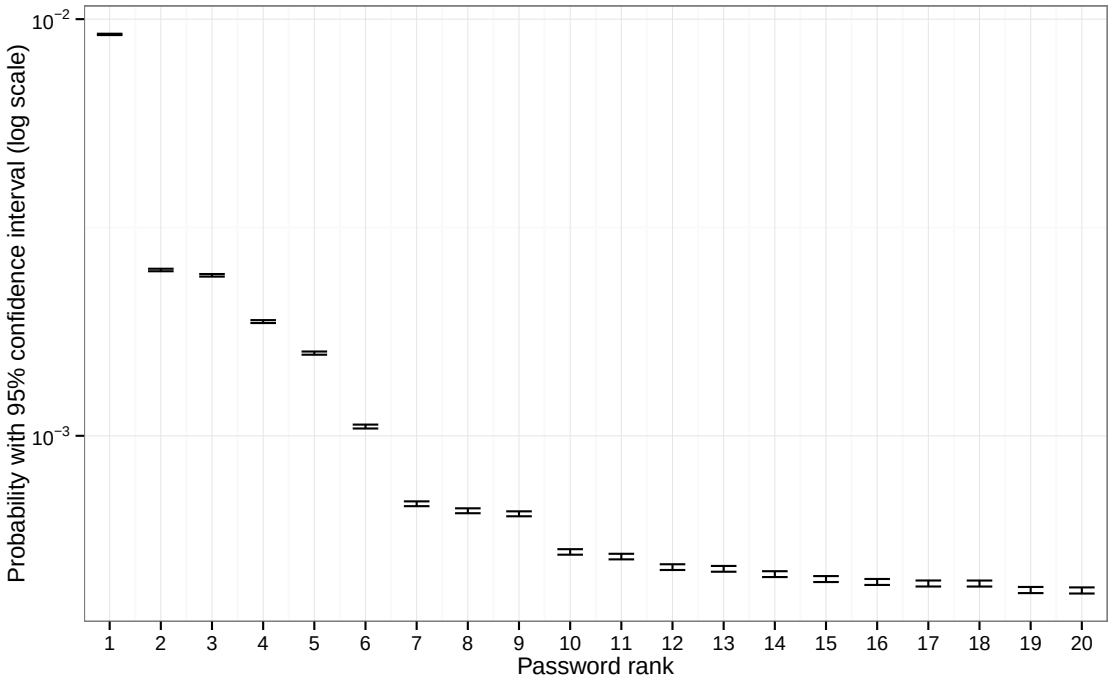


Figure 1.3: Zoom-in on the top 20 passwords from Figure 1.2. Note that as soon as we reach passwords 8 and 9 we are unsure of the true ranking of passwords by probability.

words, we do not have confidence that the eighth password ("rockyou") will truly crack more accounts than the ninth password ("12345678").

Let us ignore that issue for the moment and accept that their ordering might be incorrect. We at least have confidence that they are both more probable than the tenth password ("abc123"). We run into a new problem, however, once we reach the 193,068th entry in the list. Remember that our dataset of passwords is just a sample from a distribution. It does not represent the universe of all passwords. Consider all the passwords that are not in our sample. If we could collect one more password from this distribution, there is a 36.5% chance that it would be a password we have not seen before.[5] The 14,151,104 lowest probability passwords in our sample all have confidence intervals that overlap. The confidence interval for our new password would also overlap with these passwords. Therefore, we cannot rule out all of the passwords we have not seen yet from our list. In other words, we cannot make more than 193,067 guesses with 95% statistical confidence. At a 75% confidence level, we reach this point after 328,013 guesses.

[5]This is based on a concept called simple Good-Turing estimation [51]. We have 31,630,845 total passwords and 11,884,429 appear only once in the sample: $\frac{11884429}{31630845} = 36.5\%$.

The takeaway from this exercise is that accurately characterizing password distributions requires more samples than are feasible to collect. This issue is discussed in more depth in Section 2.5.1. The current state of the art, which relies on probability estimates, suffers from a lack of data when considering attacks that make more than a handful of guesses.[6] This lack of data hampers the selection of new password policies. When evaluating new policies for the purpose of selecting one that might increase security, it is not feasible to collect statistics on millions of passwords, so there is a need for metrics that are applicable to relatively small samples. Few metrics of this type have been published previously, and many researchers find those that have been published, such as entropy heuristics, unsatisfying [6, 145].

1.3 Thesis statement

The goal of this thesis is to provide an improved metric for evaluating password policies, to help administrators make better policy decisions. We claim that:

> Our automated approach to evaluating password-composition policies can model a more efficient and more powerful adversary than previous machine-learning approaches, assuming an offline-attack threat model.

We accomplish this by modeling guesses using a formal grammar that can assign probabilities to passwords, even ones that we have not seen before. In the process, we build heavily on the work of Weir et al. who previously modeled passwords using a probabilistic context-free grammar [146].

An advantage of machine-learning approaches, over those that rely on probability estimates, is that we can see inside passwords to find common structures. A password-probability distribution is a list of opaque items (passwords) and their associated probabilities. In contrast, a formal grammar is a much richer model of password construction.

We also believe that a machine-learning approach can be realistic. It is reasonable to assume that adversaries also lack knowledge of the true distribution of passwords, given the amount of data that would be required to gain such knowledge, but they still attempt to guess large numbers of passwords as accurately as possible given the data available. It is known that professional, or "white-hat," crackers incorporate experience and intuition into their attacks, manually identifying patterns in passwords in order to crack them more

[6] Note that probability is still the ideal metric if one is concerned with a small number of guesses and enough data is available to get reasonable probability estimates.

efficiently [54]. This thesis, in contrast to a manual approach, uses machine learning to learn password patterns automatically. We hope that this provides a framework for studying passwords in a more principled way that is also more reproducible than manual techniques.

Our approach can also partially address the data issue in two ways. First, following Weir et al. [146], we supplement password data with data from other sources. For example, we include all alphabetic strings from the Google Web Corpus [15] as an input to our grammar. This allows the grammar to insert words from this large corpus in place of words of the same length in passwords. Second, we modify the grammar so that it can produce *unseen* terminals, strings that have not been seen before.

That said, there are plenty of sources of error that can affect our approach. The automatic nature of our framework might not model true adversaries well, and selecting appropriate training data is crucial for the accuracy of our models. Even with perfect training data, we can only provide a lower-bound on the strength of adversaries—manual techniques, or other techniques outside the scope of this thesis, might find that adversaries can be much more powerful than we think. We show that our approach can crack more passwords than previous PCFG modeling techniques, but we do not know how much further there is to go. In Section 8.2, we discuss many limitations of our approach.

1.4 Contributions

The contributions of this thesis can be grouped into three areas. Section 1.6 contains an outline of this thesis that shows where each of these contributions are presented.

The guess-calculator framework We develop a system that automatically learns a guessing model for passwords, based on a configurable mix of training data, and applies the model to a plaintext test set. It does this using a process that can be significantly faster than generating guesses explicitly.

The model learned is Weir's probabilistic context-free grammar (PCFG) [146] with some modifications. Our implementation takes advantage of unique features of the grammar to evaluate passwords in a manner that operates much faster than explicitly enumerating individual guesses—sometimes 100 times faster on the same hardware. In addition, it can handle grammars that include huge lists of strings, such as lists of multiword phrases (such as [15]).

Improvements to the guessing model We make several improvements to Weir's PCFG to model a more sophisticated adversary, increasing the guessing efficiency of our models.

These improvements include learning probabilities for strings, generating unseen terminals, linguistic tokenization, and hybrid structures. We also present a negative result with an unsupervised-tokenization approach, though we do not attempt a thorough examination of this topic.

Methodological guidance We provide guidance on the analysis of data collected using the guess-calculator framework, and start-to-finish examples of its use in the form of case studies.

The guidance includes an examination of existing statistical methods, including methods from survival analysis, and their use in comparing guessing results from different policies. We show how the improvements we have made to the calculator affect the evaluation of policies, finding that we can produce significantly better guessing models than our previous methods for a wide range of policies. We also provide a number of case studies that show how the calculator should be configured and how evaluations can be performed. We provide a total of four case studies: leaked datasets, eight-character policies, a long-password policy, and the application of survival analysis to password data.

1.5 Previous work

Some of our contributions appeared in papers previously published by our research group. In this section, I list these papers and explain how they relate to this thesis. In particular, I list contributions that were part of the development of this thesis.

[67] Kelley, P. G., Komanduri, S., Mazurek, M. L., Shay, R., Vidas, T., Bauer, L., Christin, N., Cranor, L. F., and Lopez, J. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, Oakland '12, IEEE Computer Society (Washington, DC, USA, 2012), 523–537.

This paper introduces the guess-number calculator, an early version of the guess-calculator framework, built on "Weir" and "brute-force Markov" models. These were two separate systems. I built the calculator based on the Weir guessing model, and this included many special-purpose algorithms I developed that are described in Chapter 4. I also introduced the uppercase nonterminal in this paper, described in Section 5.4.1, which was suggested by discussions within our research group.

The paper also presents experiments on the selection and weighting of training and test data that I conducted. We revisit some of these experiments with an improved framework in Section 6.2 as part of our methodological guidance.

[95] Mazurek, M. L., Komanduri, S., Vidas, T., Bauer, L., Christin, N., Cranor, L. F., Kelley, P. G., Shay, R., and Ur, B. Measuring password guessability for an entire university. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, ACM (New York, NY, USA, 2013), 173–186.

In this paper, we conducted an evaluation of users' passwords at Carnegie Mellon University (CMU). For security, this evaluation was conducted on machines to which we did not have access. Therefore, we developed a more mature version of the guess-number calculator that can run experiments in an automated and independent way. I contributed to the development of this calculator, a further step toward the guess-calculator framework described in Chapter 4. Other members of our group: Michelle L. Mazurek and Timothy Vidas, also contributed to the guess-number calculator at this time, and many of their design decisions influenced the current version of the guess-calculator framework.

The paper also introduces the use of survival analysis in passwords research. I developed the methodology for this analysis. This methodology is a contribution of this thesis and is described in Section 6.1.4. We did not invent any new statistical techniques, but we provide guidance on selecting among several available techniques.

Finally, the paper presents results on correlations between demographic and behavioral variables and password strength. It also compares CMU users' passwords to those from leaked datasets. Both of those results are presented in this thesis as a case study in Section 7.4.

[128] Shay, R., Komanduri, S., Durity, A. L., Huh, P. S., Mazurek, M. L., Segreti, S. M., Ur, B., Bauer, L., Christin, N., and Cranor, L. F. Can long passwords be secure and usable? In *Proceedings of the 2014 Annual ACM Conference on Human Factors in Computing Systems*, CHI '14, ACM (New York, NY, USA, 2014), 2927–2936.

This paper evaluates several password-composition policies with passwords of twelve characters or more. We do not claim any of the results of this paper as contributions to this thesis. Rather, I made two improvements to the Weir guessing model for this paper: linguistic tokenization (Section 5.5.2) and learning alphabetic

string frequencies (Section 5.2). This thesis presents these improvements in much more detail than the paper.

[135] Ur, B., Segreti, S. M., Bauer, L., Christin, N., Cranor, L. F., Komanduri, S., Kurilova, D., Mazurek, M. L., Melicher, W., and Shay, R. Measuring real-world accuracies and biases in modeling password guessability. In *Proceedings of the 24th USENIX conference on Security symposium*, USENIX Security '15, USENIX Association (Washington, D.C., Aug. 2015), 463–481

This paper explores different cracking tools and compares them with results from a professional cracking firm. I present some of these results starting in Section 3.3. They are used as an external point of comparison, and the analyses including them are new, but the results are not claimed as a contribution of this thesis.

I contributed an improved version of the guess-calculator framework to this paper as well, with the *unseen terminals* improvement presented in Section 5.3. As with the other improvements to the guessing model, we present these improvements in much more detail here than in the paper.

1.6 Thesis outline

In this section, we present a high-level overview of each thesis chapter and describe how the chapters relate to one another and the contributions of this thesis.

Chapter 1 presents our high-level goals, introduces some necessary background concepts, and discusses how the thesis chapters relate to each other and previous work. We also provide links for downloading artifacts of this thesis, such as the source code for the guess-calculator framework and configuration files for experiments.

Chapter 2 begins by discussing background concepts in more detail, such as policies and Weir's PCFG. The rest of the chapter is a survey of related work.

Chapter 3 introduces the core concepts of password-strength evaluation with a simple, automated algorithm we call *Simpleguess*. The output of this algorithm is a mapping of passwords to guess numbers that we call *guessing data*. We compare our guessing data to data provided by a professional security firm and to Weir's original PCFG method. These other methods provide an external point of comparison for our approach.

Chapter 4 is a system description for one of the contributions of this thesis: the guess-calculator framework. Our framework takes training data, test data, and configuration parameters as inputs and outputs guessing data. We explain how we improved on the usability of Weir's PCFG. We make evaluations more

time-efficient by not enumerating guesses explicitly, and we allow training data to be specified in a more fine-grained way.

Chapter 5 presents another contribution of this thesis: improvements to the guessing model. All of these improvements are aimed toward producing greater guessing efficiency, and succeed at doing so in many situations. This chapter presents many results showing the incremental increase in guessing efficiency produced by each improvement.

Chapter 6 aims at providing methodological and statistical guidance, to help researchers perform and analyze password-strength evaluations. The first half of the chapter describes various useful metrics identified by Bonneau [7] for use with probability distributions, and provides formulas for computing these metrics given guessing data. It stands relatively independent from the rest of the thesis. The second half of the chapter presents the results of a few experiments we performed to understand how varying inputs to the guess-calculator framework can affect its output. For example, we look at the effect of adding more training data and changing the weights of training data. There is much more work to be done in this area, and we provide only some initial results.

Chapter 7 provides case studies. This chapter, along with Chapter 6, together present the third contribution of this thesis: methodological guidance in performing password-strength evaluations. We present results, such as evaluations of long passwords and passwords with complexity requirements, that other researchers could copy and modify to perform their own evaluations. We believe these results are of general interest even to those that do not plan to use the guess-calculator framework.

Chapter 8 concludes this thesis with a summary of findings, limitations, ideas for future work, and a discussion of societal impact.

1.7 Artifacts

We make several artifacts of this thesis available for download: the guess-calculator framework, experiment configurations, scripts used for preparing and cleaning datasets, and scripts used for performing evaluations with other guessing models such as the Simpleguess algorithm.

You can download the guess-calculator framework from `https://github.com/cupslab/guess-calculator-framework/releases/tag/v1.0.0`. We provide a list of requirements in `https://github.com/cupslab/guess-calculator-framework/blob/v1.0.0/INSTALL.md` and there are extensive usage instructions at `https://github.com/cupslab/guess-calculator-framework/blob/v1.0.`

0/USAGE.md. The framework also includes a plotting library that was used to produce most of the graphs in this thesis.

Experiment configurations are encoded in files that you can find at https://cups.cs.cmu.edu/chili/projects/passwords-gcf-thesis-configs/wiki/Wiki. Each experiment has its own set of parameters and is identified in this thesis by a name, e.g., *Experiment 1*. The repository is organized by experiment name, with one folder per experiment. It also contains those public datasets that we can distribute publicly.

We do not provide public links to many of the datasets used in our experiments, but instructions for preparing these datasets can be found in Appendix B.4. The instructions there refer to scripts that can be found at https://cups.cs.cmu.edu/chilisvn/passwords-gcf-thesis-configs/standard%20input%20files/.

Finally, we provide a few scripts for performing password-strength evaluations with other guessing models. These are provided in the same repository as the guess-calculator framework, but in a newer revision. Using these scripts requires familiarity with the guess-calculator framework. See https://github.com/cupslab/guess-calculator-framework/blob/v1.1.1/USAGE.md#6-emulating-the-guessing-model-from-weir-2010 for instructions on using the Simpleguess algorithm and https://github.com/cupslab/guess-calculator-framework/blob/v1.1.1/USAGE.md#7-using-weirs-original-pcfg-guessing-model for guidance on using Weir's original PCFG code [143] to create data that is compatible with the plotting tools provided in the guess-calculator framework.

Chapter 2

Background and Related Work

In this chapter, I define terms and introduce concepts that will be used often in this document. I also provide a review of related work.

2.1 Policies and probabilities

The probability distribution of a set of passwords can be affected by many different elements.

Password-composition policies impose constraints on users that they cannot avoid. We call these *requirements*. For example, a policy might require that passwords meet a minimum length, have a particular number of symbols, or pass a dictionary check.

The password-creation environment might constrain users as well. When creating or entering a password on a mobile device, for example, there is a usability cost associated with using special characters. It is possible to use them, but some users might prefer not to. Melicher et al. studied passwords created on mobile devices and found that mobile users include fewer special characters and uppercase letters in their passwords, compared to users on desktops and laptops [98]. The impact of such *implicit constraints* is generally unstudied, but is significant since it could produce passwords that are less secure than expected. In fact, Melicher et al. found evidence that mobile passwords are less secure than traditional passwords under an offline-attack threat model [98].

For simplicity, we call the complete set of constraints that passwords are created under a *target policy* even if this includes things like implicit constraints which are not set by the system administrator. Implicit constraints can affect the distribution of passwords, so an intelligent adversary should be expected to take advantage of them when attacking passwords.

It is also helpful to classify policies as *simple* or *complex*. Previous work focused on datasets that were created under very simple policies: policies with a 6–8 character minimum length requirement and possibly one other requirement such as a digit, symbol, or uppercase letter. Leaked password sets have given us insight into simple policies by providing us with millions of passwords, but the same cannot be said of complex policies, where this amount of data is unavailable. Measuring the strength of passwords that users create under complex policies is an open problem, as is understanding how implicit constraints might affect passwords. We focus on the former in this thesis, but the methods we present are applicable to both types of problems.

2.2 Collecting data from Mechanical Turk

In some parts of this thesis, we seek to evaluate policies that are quite far from standard password policies. For example, less than 1% of passwords in the RockYou dataset [139] have 16 characters. Evaluating a minimum-16-character policy using a non-random sample from RockYou raises concerns about ecological validity. In other words, would all users creating passwords under this policy make similar passwords to those in our non-random sample?

To mitigate these concerns, we use a methodology we developed in which passwords are collected from Amazon's Mechanical Turk (MTurk) using a hypothetical scenario. In this scenario, participants are asked to create new passwords under policies that we select, removing the need to select passwords out of a larger sample.

We recruit participants by posting HITs, which are advertised tasks, on MTurk. Participants first complete a consent form, and then are given instructions for completing the task. Each participant is paid $0.55 to create a password. We tell her that her password will be needed in a few days to complete the study and obtain another payment. If she returns, she is paid $0.70. Because we want participants to behave realistically, we give them the following instructions before they create their passwords:

> Imagine that your main email service provider has been attacked, and your account became compromised. You need to use a new password for your email account, since your old password may be known by the attackers.
>
> Because of the attack, your email service provider is also changing its password rules. Instead of choosing your own password, one will be

> assigned to you. We will ask you to use this password in a few days to log in again so it is important that you remember your new password. Please take the steps you would normally take to remember your email password and protect this password as you normally would protect the password for your email account. Please behave as you would if this were your real password!

We compared this wording to other forms of instructions, and found that passwords created with this wording were significantly stronger than those created with a wording that did not contain a hypothetical scenario [76].

After a participant sees these instructions, we present her with a password-composition policy and ask her to create a password. We do not ask her to create a username. Instead, we use her Amazon "Worker ID," a random identifier, to keep track of her in our database and pay her via MTurk.

The creation form includes a confirmation box so the password must be entered twice. In all cases, passwords are entered in a standard form with dots or asterisks obscuring the password.[1] After creating a password, the participant is given a survey about her password habits. Slightly different surveys were used at different times during data collection. An example of this survey can be found in Appendix B of [127]. Once the survey is complete, the participant must enter her password correctly to complete the first day of the task. We pay her using the MTurk API.

Note that the password-creation task is not a memory test. We ask participants to behave as they would with a real password, so we do not mind if participants write down the password or store it in a password manager. Instead, our focus is on the password itself and how well it compares with passwords used for real-world services. As shown in Section 7.4, we find that passwords collected from MTurk are reasonably similar to passwords in use at a university.

After two days, we use MTurk to notify participants that part two of the survey is available. The URLs we send out are tailored to each participant so that we can identify them without usernames. After successfully entering a password, we give the participant a final survey. On completion of this survey, she is paid a bonus payment using the MTurk API. An example of the second survey, called the "Day Two" survey, can be found in Appendix C of [127].

We use a between-subjects design, so that each participant is randomly assigned a single policy. We also use Amazon Worker IDs to prevent participants from participating more than once, even across multiple research studies.

[1] Passwords are entered in an HTML input field with type "password." The choice of dots, asterisks, or other characters that obscure the password is determined by the user's browser.

2.3 Using formal languages to model text passwords

In this thesis, we model passwords with a formal, probabilistic grammar. Our framework uses this grammar to generate guesses in order of estimated probability. In other words, guessing starts with those passwords that are assigned the highest probability by the grammar.

Passwords datasets are dominated by *singletons*, passwords that only appear once in the data set. In samples of 1,000 passwords, it is common for the incidence of singletons to be greater than 95% of the sample, and for the size of the intersection between two samples to be very small. Therefore, a useful guessing algorithm must be able to produce guesses that do not appear in its training data, otherwise low-probability passwords, such as singletons will never be guessed. Traditional password crackers achieve this with "mangling rules" [93]. These are rules that can be applied to words from an input dictionary to generate new guesses outside of the dictionary.

Mangling rules can be used to produce a large number of guesses, but they might have a very low rate of success, stemming from the lack of an underlying probability model for mangling. Both Narayanan et al. and Weir et al. proposed probabilistic models of guessing [106,146]. In these models, guesses can be assigned probability values—including guesses that do not appear in the training data. When guesses are made in probability order, these models can provide a significant advantage in guessing efficiency over traditional password crackers [146].

Some of the models used in previous work are still relatively crude, however. A first-order, character-level Markov model, as proposed by Narayanan and Shmatikov [106], will rarely generate intelligible text (for an example of this, see [57]). Higher-order Markov models have shown promise for simple policies [88], though the performance of such models on complex policies has not been reported. In contrast, probabilistic context-free grammars have been used previously with good results [146,151].

2.3.1 Probabilistic context-free grammars

The probabilistic context-free grammars (PCFGs) used by Weir et al. are a key component of this thesis. Following is a review of PCFGs, a discussion of the form of Weir's PCFG, and the introduction of two terms that are important to this thesis: *structures* and *patterns*.

A probabilistic context-free grammar can be defined as a 5-tuple of finite sets $\langle \Sigma, \mathcal{N}, \mathcal{S}, \mathcal{R}, \Theta \rangle$, where Σ is a set of terminals, $\mathcal{N}$ is a set of nonterminals, $\mathcal{S}$ is a special element of $\mathcal{N}$ called the start symbol, and $\mathcal{R}$ is a set of rules having

the form $A \to \xi$ with $A \in \mathcal{N}$ and $\xi \in (\Sigma \cup \mathcal{N})^*$. Each rule $(A \to \xi) \in \mathcal{R}$ is called a *production* of the nonterminal A, and each rule is assigned a probability $\theta_{A\to\xi} \in [0,1]$. The final element $\Theta = \{\theta_{A\to\xi} \mid (A \to \xi) \in \mathcal{R}\}$ is the set of all probabilities assigned to rules so that:

$$\forall A \in \mathcal{N}, \sum_{\{\xi|(A\to\xi)\in\mathcal{R}\}} \theta_{A\to\xi} = 1 \tag{2.3.1}$$

Equation (2.3.1) says that for each nonterminal A in $\mathcal{N}$, the sum of the probabilities of all of its productions must sum to 1. See [26] for more background on probabilistic context-free grammars.

2.3.1.1 Weir's PCFG

With the above definition in mind, we find that the PCFG used by Weir et al. is relatively simple. Define $Q = (\mathcal{N} \setminus \mathcal{S})$ as the set of nonterminals minus the start symbol. All production rules in Weir's PCFG are in one of two forms:

$$\mathcal{S} \to \mathcal{Q} \qquad \mathcal{Q} \in Q^* \tag{2.3.2}$$

$$A \to T \qquad A \in Q, T \in \Sigma \tag{2.3.3}$$

An example of a Weir PCFG is provided in the following section in Figure 2.1 on page 22. The grammar is non-recursive; aside from the start symbol, no nonterminal produces nonterminals. The start symbol only produces strings of nonterminals (Form 2.3.2), and nonterminals only produce strings of terminals (Form 2.3.3). Even in a non-recursive grammar, it is possible for a nonterminal to produce other nonterminals, but Weir's PCFG is simpler than this. In fact, Weir PCFGs can always be expressed as weighted, deterministic finite-state machines (FSMs). This indicates that they have much less expressive power than other PCFGs.

Weir PCFGs have an additional constraint. All terminals that replace a particular nonterminal are the same length, and each character in a terminal is of the same character class. Formally, we can write $Q = \{L_i, D_i, S_i\}, \forall i \in [1, M]$ for some maximum length M, where L represents alphabetic strings, D digits, S special characters, and i the string length of replacement terminals. For example, L_4 is an acceptable nonterminal in a Weir PCFG that would produce alphabetic strings of length 4: `that`, `with`, `this`, `from`, etc., and $\mathcal{S} \to L_4 D_1$ is an acceptable rule that would produce strings like: `that7`, `with1`, `this2`, `from4`, etc.

2.3.1.2 Producing guesses

We can derive a PCFG from a training corpus. For example, a training set of passwords `password!`, `password!`, `baseball!`, and `baseball123!` can be parsed to

$$
\begin{aligned}
\Sigma &: \{\text{password}, \text{baseball}, 123, !\} && \text{(terminals)} \\
\mathcal{N} &: \{L_8, D_3, S_1\} && \text{(nonterminals)} \\
\mathcal{R} &: \mathcal{S} \to L_8 S_1 && \text{(structures)} \\
&\quad\ \mathcal{S} \to L_8 D_3 S_1 \\
&\quad\ L_8 \to \text{password} && \text{(terminal productions)} \\
&\quad\ L_8 \to \text{baseball} \\
&\quad\ D_3 \to 123 \\
&\quad\ S_1 \to ! \\
\Theta &: \theta_{\mathcal{S} \to L_8 S_1} = 0.75 && \text{(probabilities)} \\
&\quad\ \theta_{\mathcal{S} \to L_8 D_3 S_1} = 0.25 \\
&\quad\ \theta_{L_8 \to \text{password}} = 0.5 \\
&\quad\ \theta_{L_8 \to \text{baseball}} = 0.5 \\
&\quad\ \theta_{D_3 \to 123} = 1.0 \\
&\quad\ \theta_{S_1 \to !} = 1.0
\end{aligned}
$$

Figure 2.1: Example PCFG of the form used by Weir et al. [146] for modeling passwords

produce the simple PCFG shown in Figure 2.1. The mapping from training corpus to PCFG is not one-to-one; there are other possible training corpora that could produce this PCFG.

Beginning with the start symbol, one can repeatedly apply the rules of a grammar until a string with only terminals is produced. The set of all strings that can be produced in this way is called a "language" [26], and each string in the language has an associated probability equal to the product of all rules' probabilities used in its production. To produce guesses using a PCFG, we want to produce strings in the language in descending order of probability. Weir et al. provide an algorithm for doing this using a priority-queue that stores potential "next" guesses [144].

Weir et al. use the term *structure* to refer to the string of nonterminals on the right-hand side of Rule (2.3.2). Our example grammar contains two structures that represent strings of length 12 and 9, $L_8 D_3 S_1$ and $L_8 S_1$. Structures are specific to Weir's PCFG and are an essential, high-level component of how we model passwords. Only guesses that match a previously learned structure can be produced by Weir's PCFG, but a single structure can produce thousands or millions of guesses.

The guesses produced by our simple PCFG from Figure 2.1 are shown in Table 2.1. This PCFG is only capable of producing four guesses. Note that guess 3 did not exist in the training data, and the probabilities of all strings, e.g., `password!`, are different from their observed probabilities in the training data. This is expected, and is a result of encoding the original dataset into a PCFG, which introduces assumptions about how strings are composed.

Note that the first two and last two pairs of guesses share the same probability and the same structure. We define a *pattern* as a representation of a group of guesses that share the same structure **and** the same probability. Representing groups of guesses as patterns allows us to store guesses in a more compact form. This fact is used in generating the *lookup table* that is an essential component of this thesis (see Section 4.3).

Guess #	Guess	Probability	Probability derivation
1	password!	0.375	$\theta_{\mathcal{S}\to L_8 S_1} \cdot \theta_{L_8\to\text{password}} \cdot \theta_{S_1\to !}$
2	baseball!	0.375	$\theta_{\mathcal{S}\to L_8 S_1} \cdot \theta_{L_8\to\text{baseball}} \cdot \theta_{S_1\to !}$
3	password123!	0.125	$\theta_{\mathcal{S}\to L_8 D_3 S_1} \cdot \theta_{L_8\to\text{password}} \cdot \theta_{D_3\to 123} \cdot \theta_{S_1\to !}$
4	baseball123!	0.125	$\theta_{\mathcal{S}\to L_8 D_3 S_1} \cdot \theta_{L_8\to\text{baseball}} \cdot \theta_{D_3\to 123} \cdot \theta_{S_1\to !}$

Table 2.1: Guesses produced by the example PCFG of Figure 2.1.

2.4 Properties of passwords

Information about the properties of passwords comes from three types of sources: analysis of leaked datasets, analysis of passwords in use at an organization, and self-reported data (surveys). In cases where the original passwords come from similar policies, all three sources tend to discover similar trends. In this section, I organize previous work on the composition of passwords based on the policy of the passwords being studied: simple or complex. We call a password-composition policy *simple* if it requires a length of 8 characters or less and at most one special character, and *complex* if its requirements are more strict.

It should be noted that one should only draw conclusions about password distributions from sets where all frequency information is retained. For example, after the LinkedIn breach, the file of hashes that was publicly released did not include the 57% of passwords that had already been cracked by the original attackers [62]. Any inferences about LinkedIn passwords made from only the released hashes would ignore the weakest and most frequent passwords in the distribution. This can lead to an underestimate of the policy's guessability.

2.4.1 Simple policies

To date, there has not been a public password leak from an organization with a complex policy, though many password sets from organizations with simple policies have been leaked to the Internet. Many of these password sets have been analyzed by researchers. As investigated by Florêncio and Herley, commercial websites often have simple password policies [47] so the abundance of data from simple policies is not unusual. Sets associated with different websites tend to have many passwords in common: password, password1, 123456, 12345789, iloveyou, princess, angel, and others [40, 61, 121].

Thirty percent of RockYou passwords consist of letters followed by numbers, and in both MySpace and RockYou, the most popular suffix digit is "1" [40, 125]. Over time, we see a trend of increasing use of numbers, and sometimes symbols, in passwords. Zviran and Haga reported on a study of users at a U.S. Department of Defense installation in 1999, where passwords had no requirements. They found an average password length of six characters, with 14% including numbers and less than 1% including symbols [152]. Surveying students in an information systems course about their email passwords in 2006, Bryant and Campbell found an average length of eight characters with around 65% using numbers and 3% using symbols [17]. In a survey of healthcare workers in 2008, Medlin et al. found an average length of seven characters with 87% using numbers and 16% including symbols [97]. And in 2010, Korkmaz and Dalkilic analyzed 2,500 plaintext passwords from a Turkish university with no composition requirements and found that 73% included at least one digit, while only 1% included at least one symbol [79].

In an attempt to characterize the frequency distribution of passwords (ignoring the passwords themselves), Malone and Maher looked at a few leaked datasets and found that they almost follow a Zipfian distribution, but it is not truly Zipfian [89]. Similarly, Bonneau looked at the frequency distribution of 69 million Yahoo! passwords and found that he could fit a Sichel distribution to the data, but a Kolmogorov-Smirnov test rejected the hypothesis that the sample data was drawn from the modeled distribution [7]. Nevertheless, I believe it is useful to think of passwords as having a Zipfian distribution even if their true distribution is slightly different. Like Zipfian samples, password samples are dominated by low-frequency items such as singletons.

Jakobsson and Dhiman studied how dictionary words are modified and used as part of passwords. They examined "leet" transformations and found that less than 0.2% of passwords in various leaked password sets used leet substitutions [64].

2.4.2 Complex policies

In a survey of users who recently transitioned to a complex policy, our research group reported that participants often made passwords that exceeded minimum composition requirements, contrary to expectations [129]. We also found, however, that many users place required symbols, digits, and uppercase characters in predictable locations [76,129].

Zhang et al. studied how users modify passwords under a complex policy when a previous password expires, finding that up to 17% of new passwords can be cracked in under five guesses if the old password is known [151]. Common transformations included incrementing a number, replacing one symbol with another, and moving digits from the end to the beginning of the password. The authors also examined single-character leet transformations, but do not report the proportion of users that employ them.

2.4.3 Linguistic elements

Many analyses of passwords have found that most passwords contain linguistic elements [22,97,123,152]. Names, dictionary words, place names, and keyboard patterns have been found in password sets [61, 63]. Perhaps explaining the prevalence of linguistic elements in passwords, Carstens et al. found that passwords constructed from meaningful "chunks" were more memorable than passwords constructed using more traditional advice [25].

Bonneau and Shutova examined Amazon payphrases by taking advantage of the fact that users are restricted from selecting a payphrase that has already been selected by another user [11]. This allowed for querying the set of selected payphrases. The policy required two or more words and no digits or symbols, which likely encourages linguistic elements. They found that many users selected noun bigrams found in the British National Corpus and the Google Web Corpus, and that movie and book titles provided effective guessing dictionaries.

2.4.3.1 Tokenizing

Tokenizing, as used in this thesis, is the process of breaking a password into common elements. The particular application to passwords has not been studied, but related work exists in the field of natural language processing. Tokenizing has been studied in the context of Web URLs [29], segmentation of Chinese text [83], and breaking of compound words for machine translation [16]. The most common approach is to develop a language model from existing data, where words are separated by spaces, and apply this model to find likely splits in unseparated text.

A similar approach is applied in this thesis, largely influenced by the tools made available by Wang et al. [142] who base their models on data from the Bing web crawler.

A downside of using an existing language model for tokenizing is that passwords can have tokens not typically found in natural language. For example, we have found that "p@$$" is a common token, but it would not be found in a typical language model. Liang and Klein introduced an algorithm for unsupervised word segmentation that was subsequently improved in Berg-Kirkpatrick et al. [3, 85]. This algorithm is designed to determine word boundaries from unbroken sentences, given no ground truth about the underlying vocabulary of the language. We investigate the usefulness of this approach to password research in Section 5.5.3.

2.4.4 Policy names

We follow a convention, begun in previous papers [76], for naming password-composition policies in an abbreviated way. For example, `basic8` defines a simple policy with only an 8-character length requirement, and `basic6` defines a policy with only a 6-character length requirement. The abbreviated name always ends in a number that identifies the minimum length allowed by the policy. There are two common policy families discussed in this thesis: `basic` policies, such as `basic8`, where the only requirement is a minimum length, and n`class` policies, such as `3class12`, where a password must contain at least 3 character classes and meet a minimum-length requirement of 12. In n`class` policies, the set of character classes is always lowercase alphabetic ASCII characters, uppercase alphabetic ASCII characters, digits, and symbols, where symbols are any other allowed character not within the previous three classes. A `4class` policy must contain at least one character from each of the four character classes.

In addition, we sometimes discuss two other policies: `comp8` and `andrew8`. These two policies share the requirements of `4class8` but have an additional dictionary check requirement. The dictionary check is case-insensitive and applied to a password after all non-alphabetic characters have been removed.[2] The `andrew8` policy mirrors the password-composition policy in use at Carnegie Mellon University from 2010 onwards, and uses a dictionary of 241,497 words provided to

[2]This is just one way to apply a dictionary check. One could check all contiguous alphabetic sequences within a password, or one could check the whole password against a dictionary of common passwords. The effect of dictionary checks on the strength of passwords is outside the scope of this thesis.

us by the CMU Information Security Office.[3] The `comp8` policy also mirrors the CMU policy, but used a much larger dictionary of 2,977,223 words from the freely available wordlist at `http://download.openwall.net/pub/wordlists/all.gz`.[4] The `comp8` policy was tested before the `andrew8` policy, before we requested the dictionary from the Information Security Office.

2.5 Security metrics

The most comprehensive study of password strength metrics thus far was conducted by Joseph Bonneau for his Ph.D. thesis published in 2012 [6]. Bonneau also proposes several new metrics, and a brief survey of these and other relevant metrics follows.

2.5.1 Password distributions and entropy

A common metric for key strength used in security research is entropy, a concept from information theory [126], but applying this concept to passwords has been problematic. An accurate measure of entropy requires knowledge of the entire probability distribution of passwords. Since the distribution has a very heavy tail, its true distribution is unknown—Bonneau found, after collecting about 69 million passwords (the largest password collection analyzed to date) 42.5% were unique in the dataset [7]. Standard entropy estimates produced from a sample that incompletely reflects the distribution, even a sample of 69 million, are biased toward an underestimate of the true entropy [24]. Further, when Bonneau fit a parametric model to the frequency distribution of the data, this model was significantly different from the observed distribution [7]. In other words, even a complex model with several parameters is unable to model the shape of a password-probability distribution. A large number of samples must be collected to measure the distribution accurately.

With respect to entropy, Carlton suggests that the number of samples required to get an accurate estimate of entropy is related to the smallest probability in the underlying distribution. If the smallest observed probability is p_k, then $N \gg \frac{1}{p_k}$ is required [24]. Paninski quantified the required number of samples more rigorously,

[3] This dictionary can be downloaded from `https://cups.cs.cmu.edu/chili/projects/passwords-gcf-thesis-configs/repository/raw/dictionaries/andrew8_dict.txt`.

[4] Since the dictionary check is case-insensitive and alphabetic, all words were lowercased and only alphabetic words were included in the dictionary. This wordlist has since been updated and contains many more words. The original wordlist we used can be downloaded from `https://cups.cs.cmu.edu/chili/projects/passwords-gcf-thesis-configs/repository/raw/dictionaries/comp8_dict.txt`.

and found that a reasonable estimate can found with a number of samples based on the number of "categories" in the underlying distribution [110]. For passwords, this would be the cardinality of the set of possible passwords. With m categories, Paninski suggests $\frac{m^2}{(\log m)^2}$ samples. In either case, this appears to be far more samples than is feasible to collect. Using Paninski's formula and Bonneau's identification of 33,895,873 distinct passwords in his sample, around four trillion samples are required. This is likely an underestimate of the space of possible passwords, making the true number of required samples higher still.

In 2010, our research group published an alternative method for estimating password entropy given a limited sample [76, 129]. Rather than estimate entropy based solely on the probability distribution of whole passwords, the algorithm measures the entropy of several features of each password and sums them to estimate the entropy of the distribution of the whole. In order to have enough samples to accurately estimate the entropy of the features, 1,000 samples from a given policy are recommended. This metric is not intended to provide an accurate measure of true password entropy. Rather, it is intended to provide an alternative metric that can be used to compare sets of passwords created under various policies with limited samples.

It should be noted that even with an accurate measurement, entropy might not be predictive of other security metrics. If the true metric of interest is guessing difficulty, for example, or a probability measure for the weakest passwords of a policy, one can posit a family of distributions in which these measures are held constant while entropy is made arbitrarily large. Bonneau discusses this issue extensively [6, 7]. It is possible that real-world password distributions are constrained in some way that makes these theoretical results inapplicable, but this question has not been thoroughly studied. Recently, our research group provided some evidence that our entropy estimate might be more closely correlated with guessing difficulty than naïve metrics based solely on policy requirements, such as those proposed by NIST [67].

2.5.2 Metrics based on password prediction

For a given password and guessing algorithm, "guessability" measures the number of guesses needed by the algorithm to guess that password. This particular definition of guessability is parametrized by the guessing algorithm, which means that very different measures of guessability can be obtained when using different algorithms.

To avoid this problem, Bonneau characterizes guessability based on an adversary who makes guesses in probability order, with perfect knowledge of the true distribution of passwords [7]. This obviates the details of the guessing algorithm. Examining the Yahoo! dataset that he collected, one estimates that after 200,000 guesses, 25% of passwords are guessed. In about 3,000,000 guesses, an adversary would guess 50% of passwords. Using Bonneau's approach, we could extrapolate the shape of a probability distribution out to trillions of passwords, and then compute the percentage of passwords cracked. However, an adversary cannot make guesses based purely on the shape of the distribution. Bonneau's approach is elegant, but it assumes that the adversary knows the true distribution of passwords and their probabilities even though we do not. If this assumption were incorrect, then we could not trust its estimates of the real-world security resulting from a given policy.

In 2009, Weir et al. presented a generative model for text passwords in which guesses are made in probability order [146]. The authors found their technique to be more effective than John the Ripper, an open-source password cracking tool that does not base its guessing on probability [39]. In a separate study, Zhang et al. found Weir's algorithm to be the most effective among the techniques they used [151]. Weir also made his tool and source code available online for other researchers [143].

Narayanan and Shmatikov introduced a password-cracking technique based on Markov models in 2005 [106]. While each guess of this algorithm is assigned a probability, the paper does not present an algorithm for producing guesses in probability order. However, Marechal [92] and Weir [144] both examine this model and find it more effective than John the Ripper. Further, the paper introduces the idea of quantizing probabilities, which the authors call "discretizing," to reduce memory usage at the cost of accurate password probabilities. We use quantization for a similar purpose in this thesis.

van Oorschot et al. have also studied password prediction in the context of graphical passwords. They introduce the idea of "weak password subspaces": out of all possible passwords, users typically choose passwords from a small subset of the total space [137]. They then attack password prediction from two angles: by constructing attacks based on "heuristics" which are educated guesses about what types of passwords users will choose [136], and by learning from a small sample of user passwords using a first-order Markov model [138].

This thesis primarily adopts the latter approach of learning primarily from samples of user passwords, but allows the user of our framework to include other data sources if desired. For example, we typically include dictionaries of English

text as inputs to our algorithm, because we believe that users are likely to choose words from these dictionaries even if our samples do not contain them. In this way, we provide support for policy evaluators who want to incorporate their own heuristics for how users will choose passwords into the guessing model.

It is also possible to generate guesses in a wholly different way, by collecting guesses from real adversaries using honeypots [120]. This approach might be used to find additional parts of the password space that are underrepresented in a guessing model. We do not consider this approach in this thesis, but data collected in this way would be useful for validating our guessing model.

2.6 Methodology of password strength measurement

Having a good metric is not sufficient to estimate the guessability of passwords accurately. In this section, I discuss work related to methodological issues of password strength measurement. The methodology used to collect data used in this thesis is described in Section 2.2.

2.6.1 Machine learning methods

The basic approach used in this thesis for estimating the guessability of a policy is a subjective Bayesian approach [46]. In other words, policy evaluators are able to combine samples of passwords with other data sources that they believe will improve the guessing model. It is not feasible to collect enough samples from a complex policy to estimate guessability using previous methods, but one can still generate estimates using a subjective Bayesian approach. The new data (samples from the target policy) and prior data (other data sources) can be weighted as desired based on how informative the prior data is expected to be, or the appropriate weights could be selected based on the performance of the resulting model.

Methodological issues in model generation and estimation have been studied extensively in the machine learning literature [5, 44, 56, 75]. The best practices include:

Holdout data
Separate test data (on which the quantity is estimated) and training data (on which the model is trained). If additional parameters of the model need to be selected (such as weights), hold out additional training sets to prevent overfitting.

Cross validation
Rotate training and test data throughout the sample so all data is used an equal number of times in training and testing.

Randomization
Both training and test sets should match the underlying distribution as closely as possible. With password datasets, the original data might be ordered in some non-random way, e.g., chronologically or alphabetically. Naïvely partitioning data of this form into training and test sets can produce samples that do not accurately reflect the underlying distribution.

2.6.2 Data collection and representative samples

A proper study of password policies requires a randomized experiment, in which participants are randomly assigned a password policy. Organizations could conduct such experiments, but it does not appear that an experiment of this type has been reported. Instead, password researchers have turned to conducting user studies in which participants create passwords used for the duration of the study.

However, this can create concerns of ecological validity. As we said in our 2011 paper "Of Passwords and People" [76]:

> It is difficult to demonstrate ecological validity in any password study where participants are aware they are creating a password for a study, rather than for an account they value and expect to access repeatedly over time. Ideally, password studies would be conducted by collecting data on real passwords created by real users of a deployed system.

Our methodology is described in more detail in Section 2.2.

Fahl et al. conducted a study on the ecological validity of passwords by comparing passwords created by users for a study to their real passwords [45]. They find a number of differences between the two datasets. They also find that participants who self-reported behaving differently for the study than they would in real life were more likely to create unrealistic passwords. The difficulty of studying passwords used for high-value accounts has consistently limited password research, and acquiring quality password samples from a complex policy is a non-trivial task.

Researchers have made various attempts to improve the validity of passwords created for user studies. In studying a new authentication mechanism, Karlof et al. made use of a financial incentive to add value to users' accounts. They

conducted a user study involving deception where participants' compensation (from $20–$41) was stored in an account that users created for the study [66]. They recruited over 200 participants at a cost of over $4,000. This approach would be prohibitively expensive for larger-scale studies. In contrast, many user studies have asked participants to create passwords that protect simulated accounts, small monetary amounts, a chance to win an iPod in a raffle, or access to university course materials including homework and grades [30, 37, 81, 150].

Many researchers have examined the use of Mechanical Turk workers as participants in human-subjects research. Buhrmester et al. found that American workers are slightly more representative of the U.S. population than other types of internet samples, and significantly more diverse than samples used in typical lab-based studies that heavily favor college-student participants [18]. This study, and others, found that Mechanical Turk tasks with appropriate screening criteria can provide high-quality user-study data [18, 43, 72, 111, 133].

Rather than conduct a new study to collect passwords, some researchers have studied complex policies by using conforming passwords extracted from existing sets. Weir et al. used such an approach with the RockYou set of leaked passwords, taking a subset that conformed to strict character-class requirements [145]. One of the methodological contributions of this thesis is to examine the accuracy of such an approach.

2.7 Extending probabilistic context-free grammars

Some researchers have worked on extending the grammar designed by Weir et al. in various ways to improve its guessing efficiency. Veras et al. added a large number of nonterminals to Weir's PCFG to represent parts of speech, names, and many other semantic classifications [140]. In contrast, we take the ability to generate trillions of guesses as a priority, so we do not add very much complexity to our grammar and instead focus on efficient guess-number generation and the ability to generate unseen terminals.

Replacing Weir's PCFG with a more powerful class of grammars might also improve guessing efficiency. In particular, the context-freeness of the grammar might be a hindrance in modeling passwords. Different elements of a password might be related to one another, but the current grammar treats these elements as independent. Researchers in computational linguistics have examined this flaw of PCFGs and various approaches have been developed to mitigate it. Early efforts in extending context-free grammars focused on context-sensitive grammars, more powerful grammars that can easily support the inclusion of contextual

information [65]. However, parsing the strings of a context-sensitive language is computationally expensive, so researchers have moved on to extensions of PCFGs that still contain context-free rules yet can capture dependencies between elements. These approaches are broadly called "lexicalized" grammars [33]. By adding a large number of non-terminals to the grammar, one for each tuple of dependent elements, dependencies can be captured while parsing remains computationally tractable. The most promising recent work in this field is the work of Petrov et al. [115,116] called a Hierarchical State-Split Probabilistic Context-Free Grammar (HSSPCFG). HSSPCFGs use an iterative algorithm to steadily increase the number of non-terminals in the grammar until accuracy no longer improves.

Unfortunately, there is a quite a distance between these grammars and Weir's PCFG. Research in computational linguistics works at the sentence level, with the individual tokens representing words. In passwords, we do not know where the spaces are between words, so the tokenization problem needs to be tackled before a more advanced grammar can be applied.

Chapter 3

Evaluating Password Strength with Guess Numbers

In this chapter, we introduce methods for evaluating password strength and contrast two types of approaches: automated evaluations that anyone can perform, and a manual approach performed by a professional security firm. We compare these approaches informally using "guessing curves," which are introduced here and are used throughout this thesis. Most importantly, we motivate our work by showing that Weir's PCFG model shows promise as an automated approach when applied to complex policies.

3.1 The Simpleguess algorithm

The goal of this thesis is to provide an improved metric for password strength over previous machine-learning approaches. Specifically, we improve upon the probabilistic context-free grammar approach introduced by Weir et al. in 2009 [146]. However, the simplest way to guess passwords is to avoid complex algorithms altogether and learn just the passwords in the training data. This approach was taken by Weir et al. in 2010 [145], who found it to be very successful. Since the number of guesses made is limited by the available training data, we can only generate a small number of guesses. Even so, it is still quite useful to compare this approach to other approaches that can generate many more guesses.

We call this approach *Simpleguess* and an algorithm for it is given in Algorithm S below. We are far from the first to suggest this approach, but it was not named by Weir et al. and details of the algorithm were not provided [145]. We do not claim our algorithm is optimal, but it is reasonably fast for the small data sets we

use. We also make our implementation publicly available,[1] where it can be used to construct plots like those in this thesis, such as Figure 3.1 on page 38.

Presentation We present our algorithms in the style of Knuth from *The Art of Computer Programming*. Our goal is to describe our algorithms in enough detail that one could implement them based solely on our description. We hope that the syntax we use is obvious, but if not, see page 2 of [73] for a detailed explanation. Algorithms are labeled with single letters. When an algorithm is referred to in another section, we use either letter and section number, such as Algorithm 3.1S, or a sufficiently descriptive name, such as *Simpleguess*.

Algorithm S (*Simpleguess*). Given lists of training passwords `P` and test passwords `T`, output guess numbers for each password in `T`. Guess numbers are determined by a probability-based ranking of passwords in `P`. If a password does not exist in `P`, a guess number cannot be assigned.

This algorithm makes use of hash tables denoted `FREQ` and `TEST` that are both initially empty.

S1. [Tabulate frequencies in `P`.] For each password `p` in `P`, check `FREQ[p]`. If it exists, increment its value by 1. If not, set `FREQ[p]` $\leftarrow 1$. (`FREQ` will contain all passwords in `P` and their frequencies. The number of entries in `FREQ` is equal to the number of distinct passwords in `P`.)

S2. [Sort by frequency.] Construct a new list, `SORTEDGUESS`, with two columns: password and frequency. The list contains each password in `FREQ` and is sorted by frequency in decreasing order. Ties may be broken arbitrarily. (The most frequent passwords in `P` are at the beginning of `SORTEDGUESS`, and the least frequent are at the end.)

S3. [Tabulate frequencies in `T`.] Perform the procedure from step S1 with `T`, storing frequencies in `TEST`. (`TEST` will contain all passwords in `T` and their frequencies.)

S4. [Number rows.] Number the rows in `SORTEDGUESS` starting with 1 on the first row and incrementing on each following row. These numbers are called *guess numbers*. (The most frequent password from `P` has guess number 1 because it is the first guess the model will try. The second-most-frequent password has guess number 2, and so on. The actual frequencies in the list can now be discarded.)

S5. [Assign guess numbers where possible.] Traverse `SORTEDGUESS` in order. For each password `p` and guess number `g`, check `TEST[p]`. If it exists, output

[1] Download the repository at `https://github.com/cupslab/guess-calculator-framework/tree/1.1` and follow the instructions at `https://github.com/cupslab/guess-calculator-framework/blob/1.1/USAGE.md#6-emulating-the-guessing-model-from-weir-2010`.

(p, g) TEST[p] times and then delete p from TEST.
For example, if p = "x", g = 7, and TEST[p] = 2, output ("x", 7), ("x", 7) since "x" appeared twice in T and has guess number 7.
(The only entries remaining in TEST will be those with no matching entry in SORTEDGUESS.)

S6. [Output remaining passwords.] For each password t in TEST, output (t, ∞) or some other indicator that the guess number cannot be assigned, e.g., a negative number. As in step S5, do this TEST[t] times. (The total number of items output by steps S5 and S6 must equal |T|.) ∎

We can use the Simpleguess algorithm to evaluate a basic8 policy.[2] We use RockYou as a source of both training and test data. Information on how we prepared this dataset is provided in Appendix B.4.3. We first filter the dataset to remove all non-basic8-compliant passwords, then randomly select 1,000 passwords and set these aside as the test data T. The remaining basic8 passwords comprise the training data P. The results of this evaluation are shown in the following section.

3.2 Guessing curves and guessing data

The output of Algorithm S is a set of *guessing data* for various policies. Guessing data contains a guess number for all cracked passwords in the test data, within some maximum number of guesses known as the *guess cutoff*. In the case of Algorithm S, the guess cutoff is the number of distinct passwords in the training data. We can use guessing data to determine the percentage of passwords that were cracked by the cutoff, as well as for any smaller threshold of guesses.

A "guessing curve" (a term we borrow from Bonneau [6]) is built directly from guessing data, and plots the percentage of passwords cracked over a range of guesses. Figure 3.1 shows the guessing curve resulting from our evaluation of the basic8 policy. We measure guesses in log scale on the x-axis and the percentage of cracked passwords on the y-axis.[3] The guess cutoff was 9,609,791 guesses.

From the guessing data, we can construct a table like Table 3.1 which tells us how many passwords were cracked for various numbers of guesses. As shown, almost 2% of passwords were cracked after 10 guesses, and over 50% of passwords

[2] Our convention for naming policies is described in Section 2.4.4.

[3] Note that this is flipped from the curves introduced by Bonneau, because we feel it is more natural to measure the adversary's actions (guesses) on the x-axis and the outcome on the y-axis. It also corresponds to an ECDF (empirical cumulative distribution function) plot of the guessing data, truncated at the guess cutoff, which can be produced by standard statistical software such as R. See https://stat.ethz.ch/R-manual/R-devel/library/stats/html/ecdf.html.

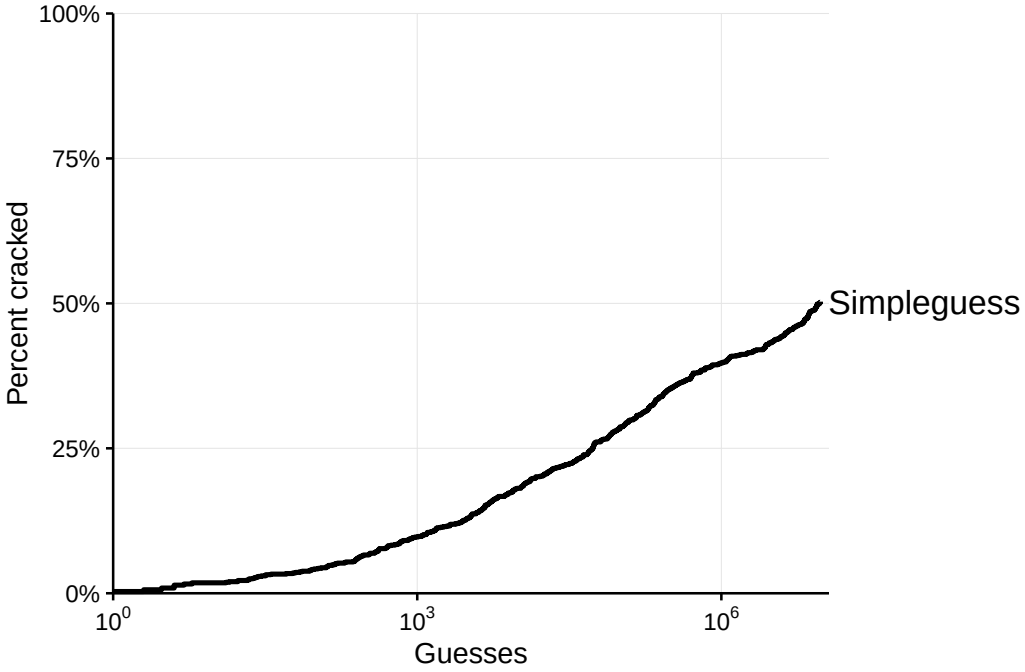


Figure 3.1: Guessing curve for a `basic8` policy using the *Simpleguess* guessing model. 1,000 passwords were randomly held out from the RockYou dataset for testing. The remainder were used for training. A total of 9,609,791 guesses were made.

Condition	10^1	10^2	10^3	10^4	10^5	10^6	Cutoff
`basic8` (Simpleguess)	1.8%	4.2%	9.7%	18.1%	28.7%	39.7%	50.3%

Table 3.1: Percent of test data cracked by the Simpleguess model at various numbers of guesses. The approach is extremely effective with the `basic8` policy. The cutoff is 9,609,791 guesses.

were cracked after about 9 million guesses. We can also determine the number of guesses needed to crack a given percentage. With this guessing data, we find that 4 guesses are needed to crack 1% of passwords, and 14 guesses are needed to crack 2% of passwords. 25% of passwords are cracked after 53,902 guesses. Chapter 6 provides guidance on how to interpret these numbers and compare sets of guessing data.

Now let us apply the Simpleguess model to a more difficult case. Consider the `3class12` policy. Only 0.6% of passwords in the RockYou dataset comply with this policy. We were concerned that such a small subset of passwords would not be representative of the passwords users would create under such a policy, so we collected `3class12` passwords from Mechanical Turk using the methodology of

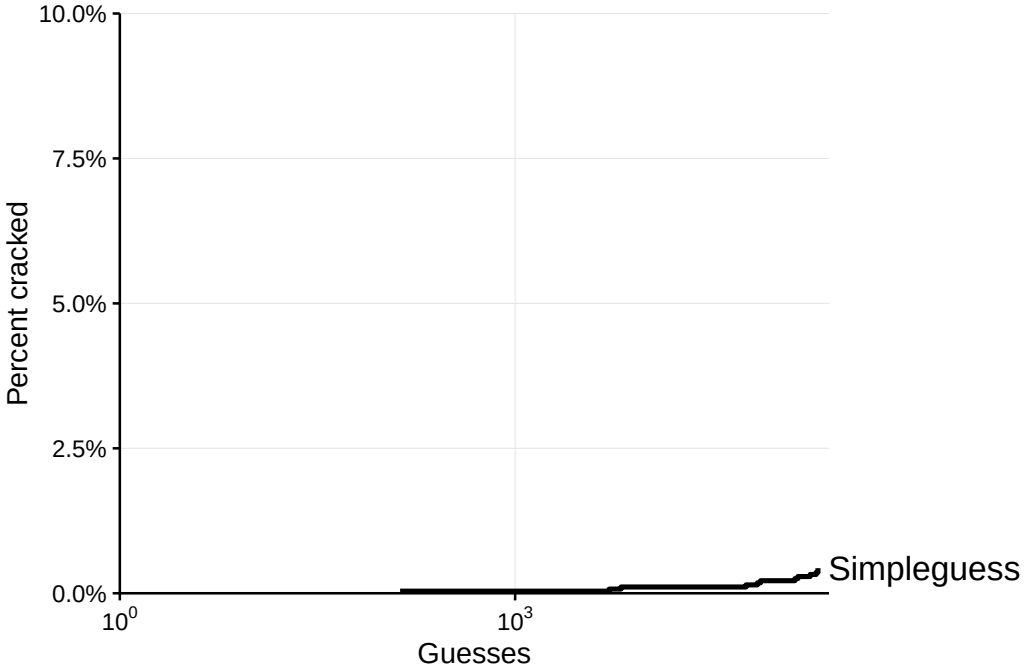


Figure 3.2: Guessing curve for a `3class12` policy using the *Simpleguess* guessing model. 2,774 passwords were collected from Mechanical Turk for testing. All `3class12` passwords from the RockYou dataset were used for training. Note the y-axis limit is at 10% cracked. The guess cutoff, i.e., the total number of guesses made by the Simpleguess algorithm, is 200,809.

Section 2.2. We used these passwords as our test data `T`. For training data, we used the subset of RockYou that complies with this policy.

Figure 3.2 shows the result of this evaluation. Less than 0.5% of the passwords in the test set were cracked! In the following sections we compare this with guessing data from professional password crackers and other automated approaches.

3.3 Professional crackers

The underlying data for this section was collected for the paper "Measuring Real-World Accuracies and Biases in Modeling Password Guessability" authored by several members of our research group [135]. My contribution to this paper was the guess-calculator framework developed in this thesis and guidance on its configuration. The analysis presented here, including the comparison with other guessing models, is my own and has not been published previously. I thank Sean M. Segreti and Blase Ur for coordinating with KoreLogic Security to collect this data.

As shown in Figure 3.2, `3class12`-compliant passwords from a large dataset like RockYou matched less than 0.5% of our test set. This was found using the Simpleguess model, where passwords from the training set are tried verbatim against passwords in the test set. There are two obvious options for improving our guessing model: gathering more training data, or modifying our learning algorithm to produce a more effective model. This thesis focuses on the latter approach.

An obvious question is: what do real-world adversaries do? We do not have a definitive answer to this question, but we know that publicly available cracking tools primarily employ "mangling" to create new passwords [54], and that professional crackers use these tools as well. Mangling is a generic term used to describe various transformations that can be applied to an input password set, such as appending random numbers or substituting letters [93]. The space of possible manglings is huge, and guidance is sparse and of dubious merit, so it is difficult for a researcher to know which manglings should be applied in what order.

Rather than attempt to explore this space ourselves, members of the CMU passwords-research group coordinated with a professional security consulting firm, KoreLogic Security. To quote from our 2015 paper [135]:

> An open question in measuring password guessability using off-the-shelf, automated tools is how these attacks compare to an experienced, real-world attacker. Such attackers manually customize and dynamically update their attacks based on a target set's characteristics and initial successful cracks. To this end, we contracted an industry leader in professional password recovery services, KoreLogic ... We believe KoreLogic is representative of expert password crackers because they have organized the DEF CON "Crack Me If You Can" password-cracking contest since 2010 [77] and perform password-recovery services for many Fortune-500 companies [78]. For this study, they instrumented their distributed cracking infrastructure to count guesses.
>
> Like most experienced crackers, the KoreLogic analysts used tools including JTR and Hashcat with proprietary wordlists, mangling rules, mask lists, and Markov models optimized over 10 years of password auditing. Similarly, they dynamically update their mangling rules (termed freestyle rules) as additional passwords are cracked.... An experienced password analyst wrote freestyle rules for each set before cracking began, and again after 10^{13} guesses based on the passwords guessed to that point. They made more than 10^{14} guesses per set.

> [basic16] and [3class12] approaches are rare in corporate environments and thus relatively unfamiliar to real-world attackers. To mitigate this unfamiliarity, we randomly split each set in two and designated half for training and half for testing. We provided analysts with the training half (in plaintext) to familiarize them with common patterns in these sets. Because we found that automated approaches can already crack most [basic8] passwords, rendering them insecure, we chose not to have the professionals attack [basic8] passwords.

KoreLogic Security were compensated for their efforts and for time used on their cracking grid. It is worth noting that the tools used by KoreLogic and other professional crackers do not normally assign guess numbers when passwords are cracked. In order to perform their evaluation, KoreLogic staff advocated strongly to the Hashcat developers that they add the ability to assign guess numbers as an optional feature.[4] This feature is now available in Hashcat version 1.20 and up. We were also given a preview release of this version of Hashcat which we used to confirm the accuracy of the guess numbers it produces.

To compare KoreLogic's results to our own, we need to restrict our test set to the same passwords that they used. This is a much smaller set than before, only 495 passwords, because we had only collected 990 3class12 passwords at the time the evaluation was performed and gave KoreLogic half of them to use as training data.

Figure 3.3 shows Simpleguess and KoreLogic's (hereafter referred to as *professionals*) results on this smaller test set. All 3class12 passwords from the RockYou dataset were used for training the Simpleguess model, while the professionals were given 495 3class12 passwords from Mechanical Turk to learn from and were permitted to use whatever additional data they wanted. Just as in Figure 3.2, the Simpleguess algorithm does not perform well. It only cracks a single password! In contrast, the professionals are able to crack over 32% of the passwords, though it takes several hundred billion guesses before they crack the first password. As we will show in the next section, a PCFG approach is able to strike a balance between these two approaches; we can crack many passwords early on while continuing to crack passwords as the number of guesses grows large.

[4] Hashcat (which is sometimes called *hashcat because of the OpenCL and CUDA varieties called oclHashcat and cudaHashcat respectively) seems to be the most popular tool for cracking password hashes using GPUs [54]. It is freely available but closed source, unlike JtR [39] which is open source, so all features must be agreed upon by a small group of developers.

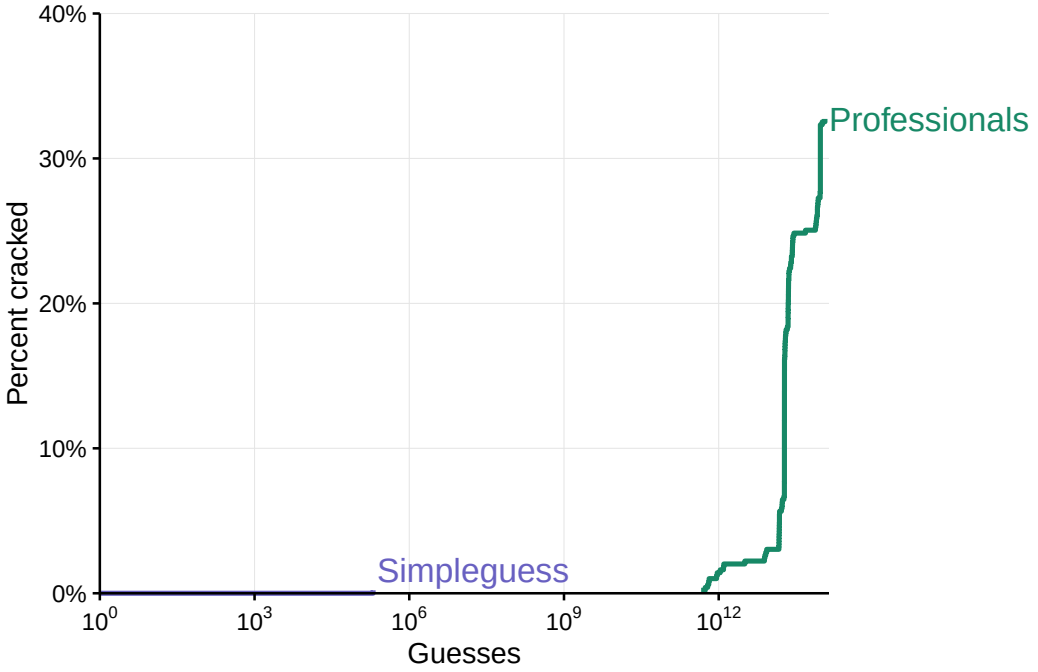


Figure 3.3: Guessing curves for the `3class12` policy for both the Simpleguess guessing model and professional crackers. The test set consists of 495 passwords collected from Mechanical Turk. The Simpleguess model only cracks one password and makes a total of 200,809 guesses. The professionals do not crack their first password until guess 456,884,372,152 but eventually crack 32.7% of passwords by guess 114,602,338,907,833.

3.4 Guessing with a PCFG

We next look at how Weir's PCFG guessing model [146] performs on the same training and test data. This model was first described in Section 2.3.1, and lays the groundwork for our approach. In this section, we describe how Weir's original implementation can be used to generate guessing data for our training and test sets.

Figure 3.4 shows a high-level overview of Weir et al.'s approach to generating guess numbers [146]. Unfortunately, the code released by Weir [143] cannot be used on arbitrary training and test data in a straightforward way. We have redistributed this code[5] along with helper scripts that allow experiments to be run easily. We do not consider this to be a major contribution of this thesis, but we believe it will help other researchers compare their approaches to Weir et al.'s. This code can be found at `https://github.com/cupslab/guess-calculator-framework/tree/v1.1.1/weir2009`.

[5]Weir's original implementation was released publicly under a GPLv2 license (`http://www.gnu.org/licenses/gpl-2.0.html`).

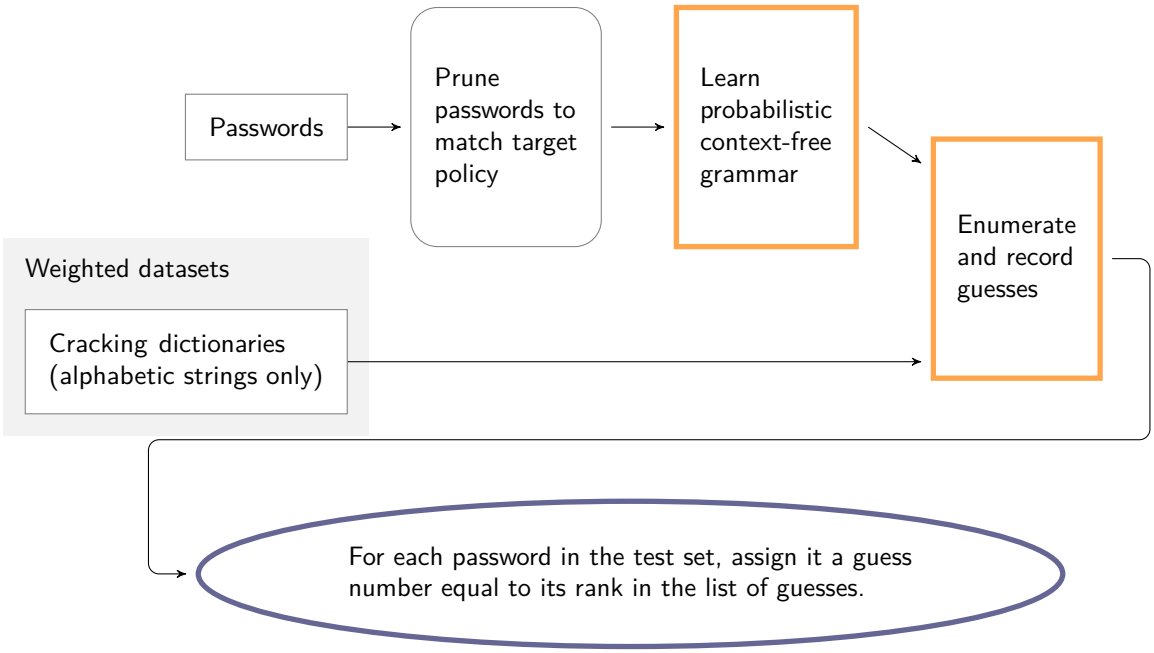


Figure 3.4: Approach to guess number generation employed by Weir et al. [146]. An explanation is provided in the text. Only steps 2 and 3 (orange boxes with sharp corners) are automated in the implementation provided by Weir [143]. We provide code for automating step 4 (blue ellipse) as well. The framework developed in this thesis automates the entire process and provides additional features. It is described in Chapters 4 and 5.

Let us describe the process illustrated by Figure 3.4 in detail. There are two sets of inputs: passwords and cracking dictionaries. Cracking dictionaries are sets of words, also called "wordlists," which are the traditional form of input for cracking tools such as JtR.[6] Independently, a PCFG is learned from those passwords that comply with the target policy. The cracking dictionaries and PCFG are used together to generate guesses, with strings from the dictionaries substituted in for alphabetic strings in the grammar. More formally, the cracking dictionaries are the only source of terminals for the alphabetic nonterminals (L_i) in the grammar. Each cracking dictionary can be assigned a weight, but string probabilities within a dictionary are ignored.[7]

Weir's implementation is notable in that it can produce guesses in probability order. It does this by maintaining a list of all potential next guesses in a priority queue. As each guess is produced, it adds that guess's successors to the queue [146].

[6] `http://www.openwall.com/wordlists/`

[7] Note that Weir's implementation does not require that the cracking dictionaries contain only alphabetic strings. However, Weir provides command-line switches to remove any non-alphabetic strings from the dictionary inputs and recommends that they be used [143].

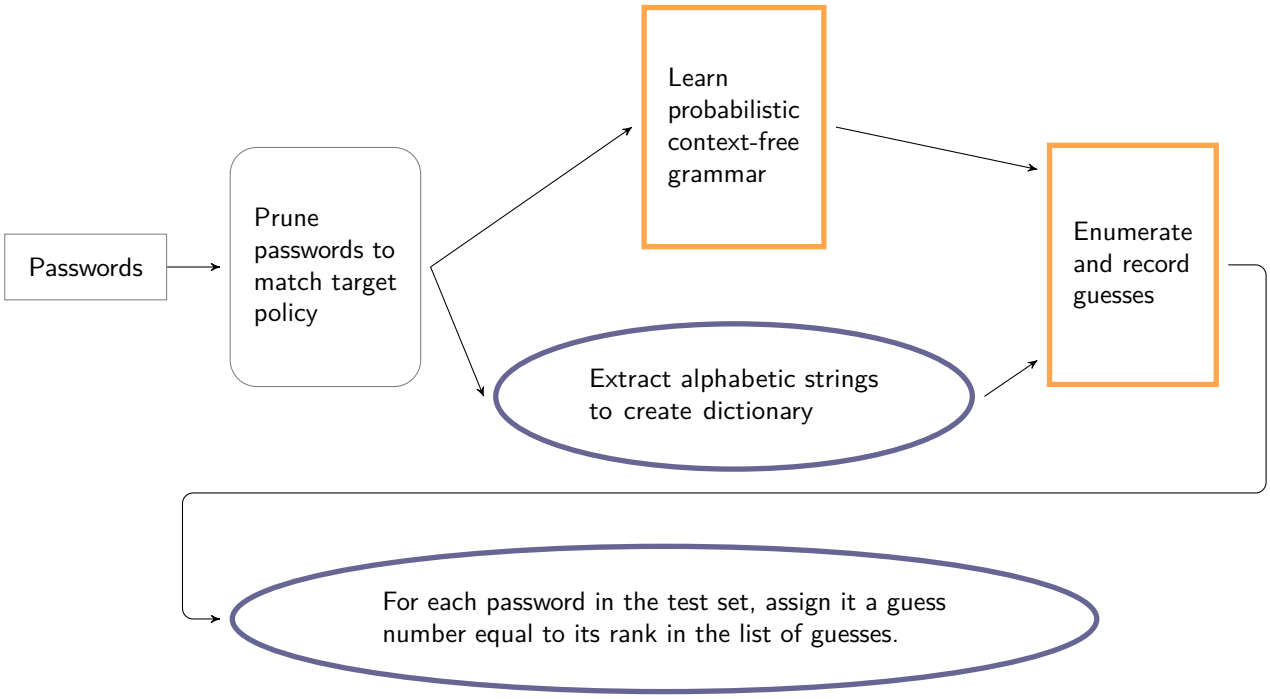


Figure 3.5: Approach to guess number generation used in this section. Compare with Figure 3.4. For simplicity, we use a single set of pruned input passwords from which we learn both a grammar and alphabetic strings. This makes it comparable with guessing models derived using the Simpleguess algorithm, which also takes a single set of input passwords. Orange boxes with a thick outline come from Weir's original implementation. Blue ellipses are provided by our implementation.

The size of the queue is unbounded and can grow to an impractically large size over time, so we provide a script for terminating the generation process when a threshold amount of free RAM on the machine is exceeded.

Once the list of sorted guesses is produced, we can use it to assign guess numbers. This operation is encapsulated in the final ellipse in Figure 3.4. We implement this with a straightforward modification of the Simpleguess algorithm on page 36: skip steps S1 and S2 and let `SORTEDGUESS` be our generated list of guesses. The remaining steps of the algorithm can then be followed without modification. The final output of the algorithm is a set of guessing data.

3.4.1 Comparison with other models

To make a comparison between Simpleguess and Weir's PCFG model more fair, we should isolate the algorithm from its training data. Our Simpleguess evaluation used a single source of training data: `3class12` passwords from the RockYou dataset. To use the same input with Weir's PCFG approach, we need to modify

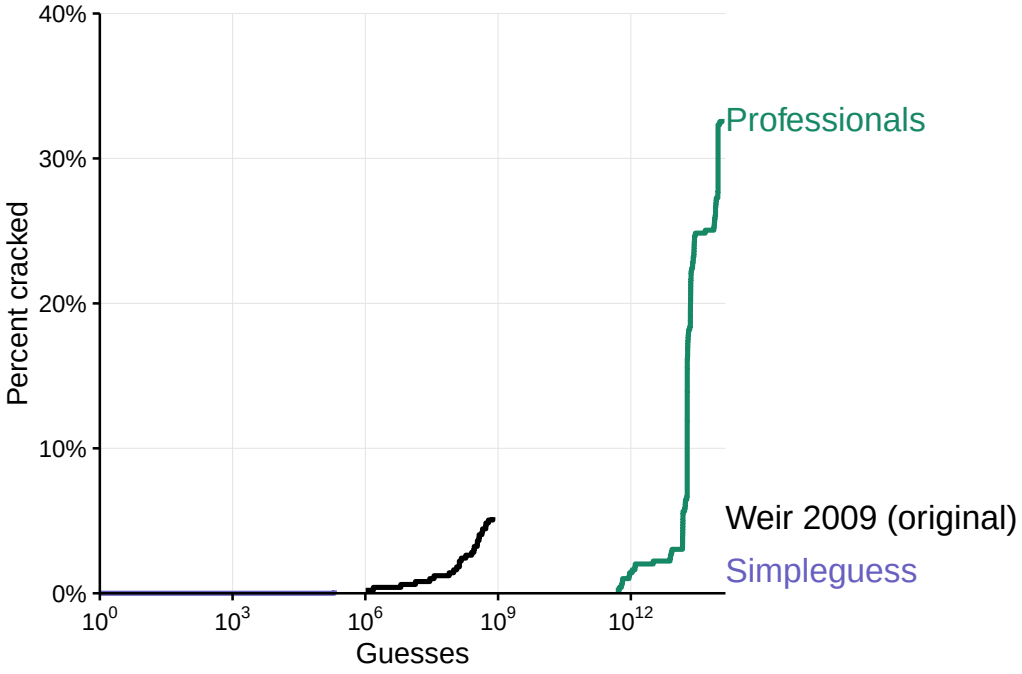


Figure 3.6: Guessing curves for the `3class12` policy generated using the Simpleguess algorithm, Weir's original 2009 PCFG implementation [146], and professional crackers. Training data, test data, and the Simpleguess and Professionals curves are the same as in Figure 3.3. Curve labels are aligned to the right of the graph to prevent labels from colliding with curves. The Weir 2009 curve represents 801,985,812 guesses and cracks 5.3% of passwords. Like Simpleguess, it is an automated approach, and uses only the `3class12` passwords from RockYou as training data.

the process in Figure 3.4 slightly so that cracking dictionaries are not needed. This is shown in Figure 3.5. We only need to replace the cracking dictionary input with alphabetic strings extracted from the password set.

Figure 3.6 shows the result of this approach applied to our training and test data, compared with the Simpleguess and professional approaches. Even in this simple example, we can begin to see the power of the PCFG model. Using an automated, machine-learning approach, Weir's PCFG model cracks passwords before the professionals and continues to crack passwords until the guess cutoff at 801,985,812 guesses. However, it suffers from two major problems. First, its first crack is at guess 1,017,168 compared to Simpleguess which cracked its first (and only) password at guess 198,686. In this thesis, we will show that this is a persistent problem with this PCFG model, and in Section 5.5.1 we will show why it happens and how it can be overcome. Second, we are not able to come anywhere close to the professionals in terms of the number of guesses made—the professionals made 114,602,338,907,833 guesses. In Chapter 4, we will show how

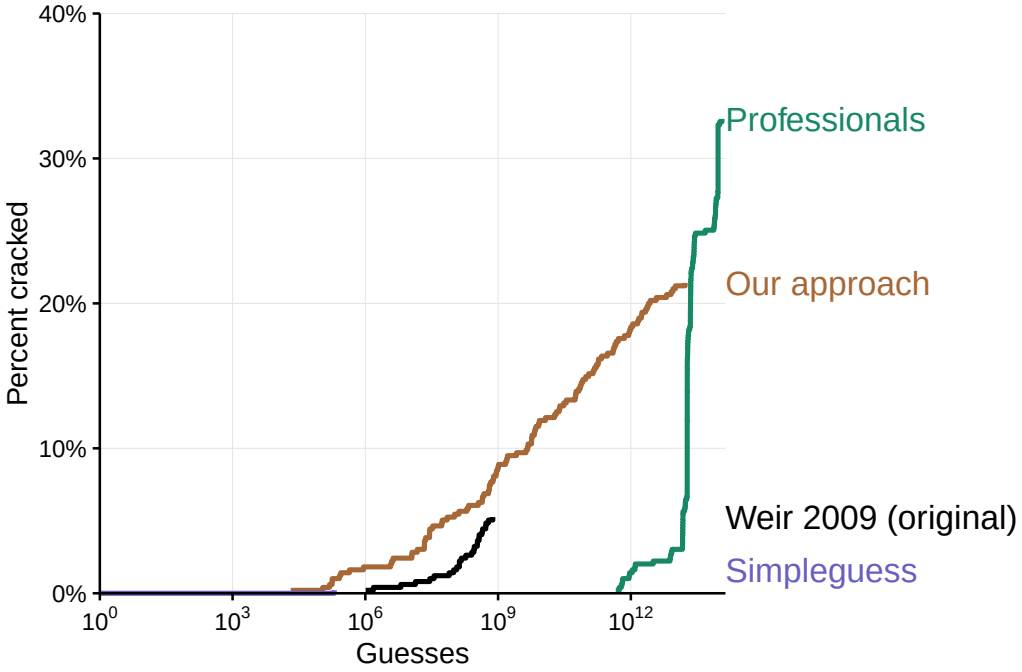


Figure 3.7: Experiment 12 — Guessing curves for the `3class12` policy generated using the Simpleguess algorithm, Weir's original 2009 PCFG implementation [146], our improvements to Weir's PCFG approach, and professional crackers. Our approach represents 16,903,290,362,610 guesses and cracks 21.4% of passwords. The configuration for this experiment is provided on page 233.

the PCFG model can be modified so that we can calculate guess numbers within an order of magnitude of this number, provided the test set is in plaintext.

As a preview of the contributions that will be presented in the next two chapters, Figure 3.7 shows how the previous guessing curves compare with the approach developed in this thesis. Our approach cracks passwords faster than any of the other approaches, including Simpleguess, though it should be noted that it takes advantage of a larger set of training data. While we do not outperform the professionals at large guess numbers, we do outperform them up to the guess cutoff for our approach at 16,903,290,362,610 guesses. We believe that this is a significant improvement over Weir's PCFG model and represents a new benchmark in automated password-strength evaluation. The configuration for our approach is provided on page 233 and will be explained in detail over the next two chapters.

Chapter 4

Efficient Guess-Number Calculation

Our tool for studying password policies is the *guess-calculator framework*, a system that calculates guess numbers for passwords of a target policy. The framework incorporates a probabilistic context-free grammar that builds on the grammar developed by Weir et al. [146]. We first introduced this system as the "guess-number calculator" in [67] where it was built using Apache Hadoop on top of a modified version of Weir's original code. In [95], we made the system more automatic and removed the dependency on Hadoop. Finally, in order to enable the improvements to the guessing model implemented in Chapter 5, we rewrote the system from scratch. However, most of the algorithms described in this chapter were developed for the early versions of the guess-number calculator.

In this chapter, we explain how we extended the methods of Weir et al. to create a framework for automatic, efficient guess-number generation. We discuss several aspects of our implementation. Most of these aspects are aimed at reducing the amount of time required to perform an evaluation, holding computational constraints constant. Some also reduce the amount of space required for intermediate processing. Other aspects make the framework more usable. For example, having the framework create a holdout set automatically prevents the researcher from having to perform this step manually. This saves researcher time and makes evaluations easier to reproduce. Finally, some aspects are aimed at increasing the richness of our models. Weir's PCFG produces a finite language and is constrained by the structures and terminals that it has learned. Our changes to how the grammar is trained enable it to produce a larger language. This can increase the guessing efficiency of the resulting model.

4.1 Improving on previous approaches

We use flowcharts to provide an overview of the steps taken by the guess-calculator framework to produce guess numbers. Figure 3.4 on page 43 showed how this is done using Weir's original implementation. Figure 4.1 on page 49 shows an early version of our framework, which illustrates the process used in many of our previous papers [67, 95, 128, 134]. Finally, Figure 4.2 shows the approach developed in this thesis. The three blue boxes correspond to the improvements to the guessing model described in Chapter 5. The rest of the boxes will be described in this chapter.

A downside of the approach of Weir et al. is the need to enumerate all guesses in probability order. This is a time-consuming process that limits the guess numbers that can be assigned. In Weir's dissertation he examines guess numbers up to 1 billion [144], but modern cracking approaches can now make several hundred billion guesses per second [54]. A further limitation of his implementation is in the quantity of training data that can be used. Due to implementation issues, Weir was only able to train the grammar on a few million passwords.

In contrast, our guess-calculator framework has been trained on hundreds of millions of input elements, and can assign guess numbers in the hundreds of trillions. In Section 3.4, we described how Weir's implementation uses a priority queue approach to generate guesses in probability order, at the cost of an ever expanding data structure. We take a very different approach, first described in Section 4.3, and expanded over the rest of the chapter. We are able to reach large guess numbers because we move the generation of guesses out of RAM and onto the disk, and employ various techniques to reduce the amount of data we need to store.

4.1.1 Implementation minutiae

The guess-number calculator was built on top of code originally released by Matt Weir under a GPLv2 license.[1] This code was entirely rewritten using an object-oriented approach, and incorporates many new features that the original code did not possess which are covered in this chapter. We also removed some implementation limitations.

Since guess numbers have the potential to overflow a long integer, the GNU Multiple Precision Arithmetic Library[2] was introduced and many integer operations are replaced with their respective operations from the library. Since we

[1] `http://www.gnu.org/licenses/gpl-2.0.html`
[2] `https://gmplib.org/`

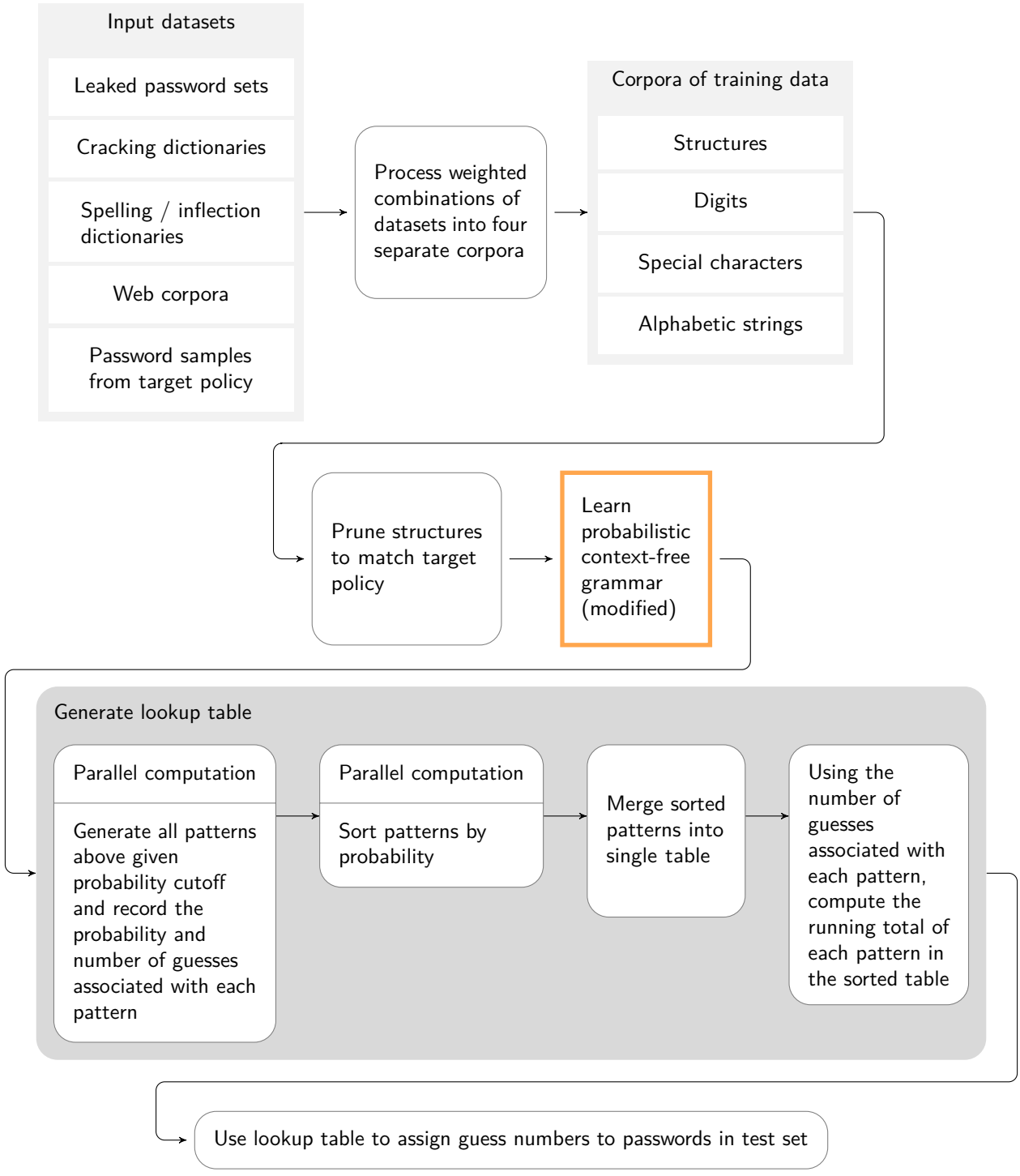


Figure 4.1: Approach to guess number generation we employed in the guess-number calculator [67], a precursor to the guess-calculator framework. The orange box with a thick outline indicates that this step was performed by a modified version of the original implementation by Weir. We developed the rest of the implementation.

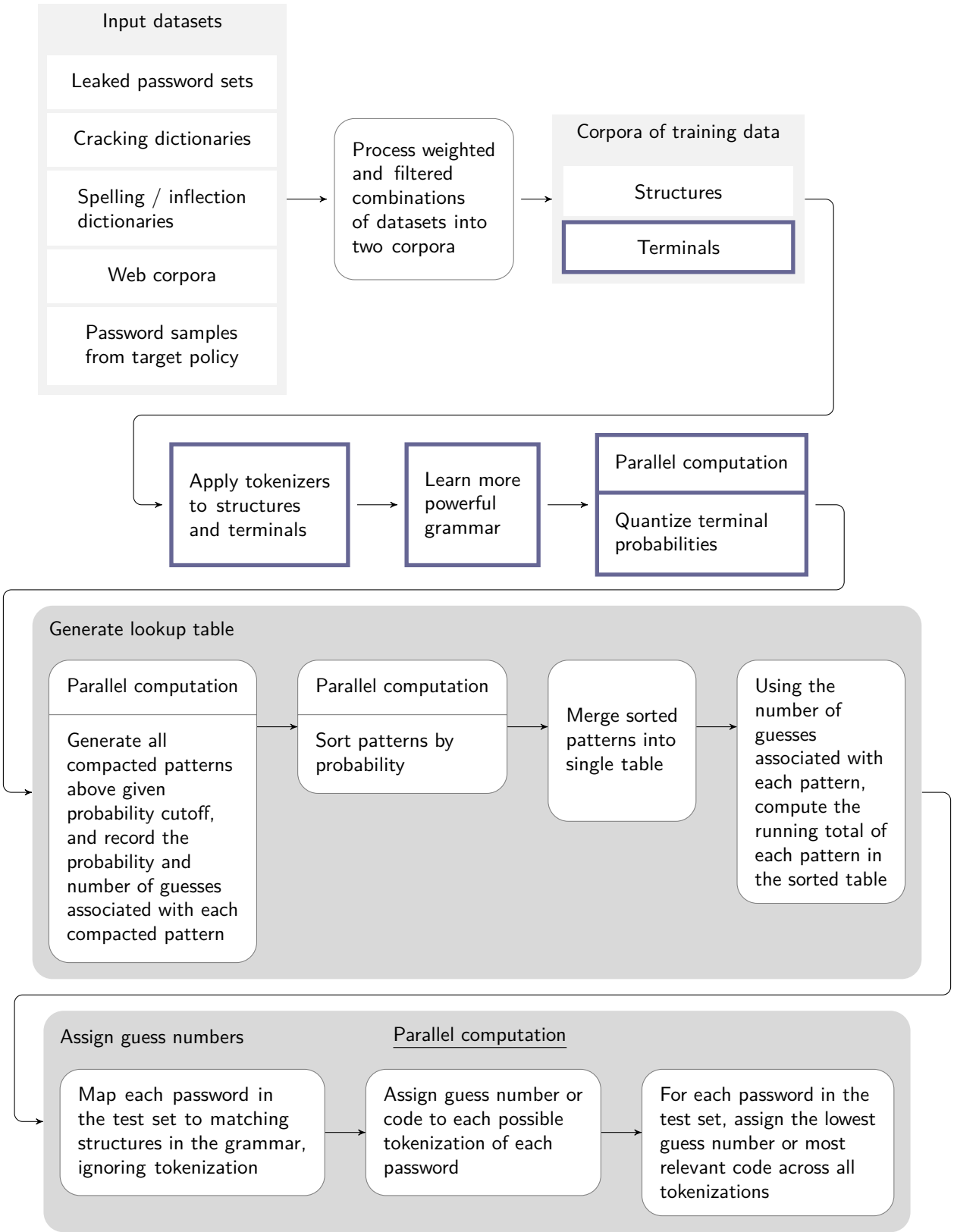


Figure 4.2: Design for the improved guess-calculator framework developed in this thesis. Many components are new or have been modified relative to Figure 4.1. Blue boxes with thick outlines are described in Chapter 5.

want to support much larger grammars than before, terminals are kept on disk and accessed via memory-mapped files, instead of being loaded into RAM. This allows for an arbitrarily large collection of terminals. Only structures and some metadata about terminals are stored in RAM.

The framework was written in C++ and many memory leaks were found and plugged with Valgrind.[3]

4.2 Richer models

In this chapter, we are only concerned with Weir PCFGs, whose main components are *terminals* and *structures*.[4] There are three types of terminals: letter strings, digit strings, and special character strings. An example structure is $L_8D_1S_1$ which describes guesses composed of an eight-character letter string, a single digit, and a single special character. For example, "password1!" is a guess that might be made under the $L_8D_1S_1$ structure.

Figure 3.4 on page 43 showed how inputs are processed to produce guesses in Weir's original approach. Only two types of inputs can be provided: a file of passwords, and "cracking dictionaries" with alphabetic strings [143]. Looking more deeply into the implementation, we find that structures, digit strings, and special character strings are all learned from the file of passwords, and the letter strings come solely from the cracking dictionaries. A notable aspect of this approach is that the cracking dictionaries can be independent of the passwords—they can come from other password sets, strings collected from the Internet, or even randomly generated strings. The only constraint is that there must be a corresponding length string for each letter nonterminal in the grammar. For example, if a string composed of 16 letters exists in the passwords input, there must be a string of 16 letters in a cracking dictionary or the program will fail.

We took this idea to its logical conclusion. As Figure 4.1 shows, we allowed four types of input to the grammar: sources of structures, digits, special characters, and alphabetic strings (letters). Input datasets were processed into four different *corpora*, and each corpus was used to produce the corresponding component of the grammar. We use the term *corpus* to refer to a single file produced from multiple input files. In Section 5.4.2, we allow the grammar to contain mixed-class terminals and nonterminals, removing the restriction that terminals must come from a single character class. To accommodate this, we no longer split the inputs across three

[3] `http://valgrind.org/`

[4] For a reminder on the form of a Weir PCFG, examine the set of production rules $\mathcal{R}$ in Figure 2.1 on page 22.

terminal types. Instead, we produce a single corpus for terminals and rely on the user to populate it as they see fit. This change can be seen in Figure 4.2. We make it easy for users to emulate the old approach and split their terminal sources into alphabetic strings, digits, and special characters, but this is not required.

Users specify training data using a configuration file. The configuration file has separate sections for structure and terminal-input datasets, which are listed by file name. Users can also specify an optional weight and filter for each dataset. Filters are executable scripts that can be used to prune an input dataset to a target policy, pull out strings of interest, or otherwise modify the input.[5] For example, a user could specify a filter that extracts only alphabetic strings from the data source. Weighting will be described in the Section 4.2.1. Our corpora record the weighted frequency for each string, and a list of identifiers of datasets where a string was seen. A string, its weighted frequency, and any identifiers comprises a single *record* in a corpus.

Our algorithm for building a corpus is straightforward. We always maintain a running corpus that is initially empty. For each dataset, convert it to a new corpus by tabulating frequencies and multiplying the frequencies by a weight if specified.[6] Optionally, attach an identifier to each record that identifies this dataset. For each record in the new corpus, check if its string matches a record in the running corpus. If so, sum the frequency values and take the union of the identifiers, then replace the record in the running corpus with these values. Else, simply add in the new record. This *merge* is complete when all records in the new corpus have been processed. The running corpus is complete when all input datasets have been processed.

Motivation

It is important that we separate structures training, which determines the password-composition policy of the guesses made by the model, from terminals. We need to be careful about what passwords we allow in our structures training data, lest we make guesses with a structure that does not match the target policy. In all cases, we perform structures training with the most representative data source, e.g. real passwords that match the target policy. In contrast, the terminals corpus can include anything that might help the model. This makes the model richer, because a model with more terminals can produce more guesses.

[5] We use filters to replace the "Prune structures to match target policy" box in Figure 4.1.

[6] We tabulate frequencies with a hash table as in step S1 of the Simpleguess algorithm on page 36, and weighting is described in Section 4.2.1.

Splitting the training data also allows us to choose different sources for different terminal types. A source of alphabetic strings, like a dictionary in the colloquial sense, will probably not be a good source for digits or special characters, but it might still be useful for filling in words that do not appear in the structures data. Pruning a password file to a target policy is often performed to prepare a datset for structures training, but this will usually throw away most of the data. We can reuse this data, however, as a source of digits and special-character terminals. This tailoring of sources to terminal types has the potential to increase the guessing efficiency of the model.

One final note: the constraint in Weir's approach still holds. For each nonterminal found in the structures corpus, there must be a corresponding terminal. To prevent users of the framework from producing invalid grammars by accident, we automatically include the structures corpus in the terminals corpus. Since structures are learned from the most representative data, we do not expect this to be undesirable behavior.

4.2.1 Weighting sources

When users specify a dataset, they can also specify an optional weight. We identify two types of weighting that a user might want, `P`-weighting and `S`-weighting. For each type, we walk through an example where that type of weighting is appropriate.

Mixing probability distributions

Consider two datasets, $\mathcal{A}$ and $\mathcal{B}$, where we believe $\mathcal{B}$ is more representative of our target policy than $\mathcal{A}$. We want $\mathcal{B}$ to have a greater influence on our final model than $\mathcal{A}$ because we think it is a better match to our target policy. Therefore, we want to attach a higher weight to $\mathcal{B}$ when we specify our training data.

However, simply multiplying the frequencies in $\mathcal{B}$ by a larger weight than $\mathcal{A}$ is not enough. This is because we build up our corpora by summing the frequencies across datasets. Assume that dataset $\mathcal{A}$ has ten million passwords. Now imagine that $\mathcal{B}$ has only ten thousand passwords. If we simply combine these corpora without weighting, $\mathcal{B}$ will have very little impact on the model, because $\mathcal{A}$ is so much larger. This is not what we want! Even with a seemingly large weight, like 100, $\mathcal{B}$ is still overwhelmed by the frequencies in $\mathcal{A}$.

What we want is to mix their probabilities, rather than their frequencies, and apply a weight to the normalized datasets. We emulate this with the `P`-weighting scheme, which adjusts the frequencies in $\mathcal{B}$ based on the ratio in size between $\mathcal{A}$

and $\mathcal{B}$. A user-specified weight of "1x" would multiply the frequencies in $\mathcal{B}$ by 1,000; $\mathcal{A}$ has ten million samples and $\mathcal{B}$ has ten thousand, so we equalize them by treating each sample in $\mathcal{B}$ as if it represented 1,000 samples.

The guess-calculator framework uses a simple algorithm to accomplish this. Let S be the sum of frequencies in the current dataset. For $\mathcal{B}$, we have $S = 10{,}000$. Compute P to be the sum of frequencies in the current running corpus. Assume that we specified $\mathcal{A}$ first without a weight, so $P = 10{,}000{,}000$. Now multiply the frequencies in the current dataset by $k * \frac{P}{S}$, where k is the user-specified weight. Since we believe that $\mathcal{B}$ is more representative than $\mathcal{A}$, we might specify a weight of "10x" or "100x," which would multiply the frequencies in $\mathcal{B}$ by $10 * \frac{P}{S} = 10{,}000$ or $100 * \frac{P}{S} = 100{,}000$. This accomplishes our goal of causing $\mathcal{B}$ to have 10 or 100 times more effect on our final model than $\mathcal{A}$. We call this `P`-weighting because the actual applied weight is based on the cumulative frequencies of all previous data sources.

Mixing similar distributions

In contrast to the previous approach, consider two datasets, $\mathcal{A}$ and $\mathcal{C}$, where we believe the two datasets come from the same distribution. We have no reason to believe $\mathcal{C}$ is more representative than $\mathcal{A}$. For example, we use two leaked datasets extensively in this thesis: RockYou and Yahoo! Voices. RockYou has over 32 million passwords, while Yahoo! Voices has around 450 thousand. We do not use `P`-weighting to combine them. Instead, we simply merge them without a multiplier. This has the effect of treating the 450 thousand passwords in the Yahoo! Voices set as if they are just another 450 thousand samples from the same distribution as RockYou. This is the default weighting scheme in the guess-calculator framework.

We call this `S`-weighting. When it is used, we simply multiply the frequencies of the current dataset by the user-specified weight. For example, if the user specifies a weight of "100," we multiply the frequency of each string in the data source by 100 before merging it with the running corpus.

Limitations

Our implementation of weighting covers what we believe to be the common case, where several datasets are combined using `S`-weighting, and then one or two datasets are heavily weighted using `P`-weighting. For example, we might want our training data to include datasets $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$, where we want $\mathcal{B}$ to have the effect of $\mathcal{A}$ and $\mathcal{C}$ combined, and $\mathcal{D}$ to have equal effect to $\mathcal{B}$ and $\mathcal{A} + \mathcal{C}$.[7] We can

[7] We use $+$ to represent combining the datasets with `S`-weighting.

accomplish this by specifying our data sources in the following order: $\mathcal{A}$: weight 1, $\mathcal{C}$: weight 1, $\mathcal{B}$: weight 1x, $\mathcal{D}$: weight 0.5x.[8]

However, there are many potential combinations of datasets that it does not cover, e.g., $\mathcal{B} + \mathcal{D}$ having an equal effect to $\mathcal{A} + \mathcal{C}$. Our interface for specifying weights is not expressive enough to cover such cases.

Weighting is an unexpectedly tricky subject, because it forces us to consider how representative we believe each dataset is. When we are unsure, we use S-weighting and let the sizes of each sample speak to how much they should affect our model. If we have confidence in a particular dataset, however, we can apply P-weighting. We examine the effect of weighting, in a limited experiment, in Section 6.2.2. We also use P-weighting with good results in Section 7.1, though we do not compare it with other weighting schemes.

4.3 Generating patterns instead of guesses

Referring to Figure 4.1, the next step after learning a PCFG is to generate a *lookup table*. In this section we describe the "Generate lookup table" boxes in Figures 4.1 and 4.2 in detail. These boxes are the same in both figures, expect that the latter system uses *compacted* patterns, which will be described in Section 4.5. The purpose of the lookup table is to provide a data structure that can efficiently store a large number of passwords in probability order, and allow one to lookup the guess number of a particular password using binary search.

First, we remove the limitation described in Section 3.4 that requires an ever-expanding priority queue in RAM to generate guesses in probability order. Instead, we generate records of guesses with probabilities, and then sort these records on disk. The total number of guesses a Weir PCFG can produce can be enormous—10^{80} strings is not unusual. Therefore, we make a crucial change to Weir's implementation: you must provide the guess-calculator framework with a *probability cutoff*. The framework generates only those records whose probability is above or equal to this cutoff.

In order to save space and time while iterating over the PCFG, we make another crucial change to Weir's implementation: we generate *patterns* instead of explicit guesses. These are the same patterns introduced in Section 2.3.1.1, and we will describe patterns in more detail in Section 4.3.1. A pattern covers multiple guesses where each guess has the same probability, and a record of each found pattern is

[8] This is unintuitive, but half of the running corpus represents $\mathcal{B}$ since it was P-weighted with $\mathcal{A} + \mathcal{C}$.

written to disk. Each record includes the probability and number of guesses for that pattern.[9]

Once all patterns have been found and recorded, we sort the records on disk in decreasing probability order.[10] This means that after sorting, the first record is the highest probability pattern, the second record is second highest, and so on.

After sorting, we number the records. Recall that in step S4 of the Simpleguess algorithm,[11] we numbered our list of sorted guesses starting at 1 and counting up to label guesses with guess numbers. We do something similar with our sorted list of patterns, though each pattern represents multiple guesses. The first record is numbered 1, and the second record is numbered $1 + a_1$ where a_1 is the number of guesses covered by the first record. The third record is numbered $1 + a_1 + a_2$ where a_2 is the number of guesses covered by the second record, and this continues for all records. This is called a "running total" or a "prefix sum" [148].

Once the running totals have been computed, we have our lookup table. The running total assigns a guess number to each record equal to the first guess number covered by that record.[12] Using the lookup table, we can exactly determine the guess number of any password that is above the probability cutoff, using a process described in Section 4.7.

Storing patterns rather than explicit guesses in the lookup table saves both time and space when generating the table. This allows us to generate tables that can assign guess numbers in the trillions and beyond, in the same amount of space that could only hold billions of guesses.

The lookup table could also be used to produce guesses explicitly in probability order, but we have not implemented this.

4.3.1 Patterns

The records in our lookup table contain *patterns* instead of explicit guesses, where a pattern represents a set of guesses that all share the same probability and the same *terminal groups* in order. In this section, we clarify what these terms mean.

Figure 4.3 presents a PCFG specification with a single structure (L_8D_3) and several terminals. When we produce guesses from this PCFG, we choose an

[9] We also use *intelligent skipping*, which will be described in Section 4.4, and *pattern compaction*, which will be be described in Section 4.5, to save even more space and time while iterating over the PCFG.

[10] This can be terabytes of data. In an earlier version of the framework we used Apache Hadoop to generate the lookup table, but we found that this was not portable and was also difficult to work with and debug.

[11] See page 36.

[12] An example of this can be seen in Figure 4.5 on page 59, and is discussed in Section 4.3.1.

Θ: Structures

$\theta_{\mathcal{S}\to L_8 D_3} = 1.0$

Alphabetic terminals (T_{L_8})	Digit terminals (T_{D_3})
$\theta_{L_8\to\text{password}} = 0.2$	$\theta_{D_3\to 123} = 0.5$
$\theta_{L_8\to\text{baseball}} = 0.2$	$\theta_{D_3\to 000} = 0.05$
$\theta_{L_8\to\text{iloveyou}} = 0.2$	$\theta_{D_3\to 111} = 0.05$
$\theta_{L_8\to\text{princess}} = 0.15$	$\theta_{D_3\to 222} = 0.05$
$\theta_{L_8\to\text{sunshine}} = 0.15$	$\theta_{D_3\to 333} = 0.05$
$\theta_{L_8\to\text{superman}} = 0.1$	$\theta_{D_3\to 444} = 0.05$
$\theta_{L_8\to\text{whatever}} = 0.1$	$\theta_{D_3\to 555} = 0.05$
	$\theta_{D_3\to 666} = 0.05$
	$\theta_{D_3\to 777} = 0.05$
	$\theta_{D_3\to 888} = 0.05$
	$\theta_{D_3\to 999} = 0.05$

Figure 4.3: Weir PCFG example with many terminals. For brevity, we show only the set of probabilities Θ, since the other elements of the grammar: terminals, nonterminals, start symbol, and production rules can all be inferred from the left-hand side of each assignment. The square brackets to the left and right of the terminals columns denote *terminal groups*. A terminal group is a set of terminals that have the same probability and are produced by the same nonterminal (L_8 or D_3 in this case).

alphabetic terminal from the list T_{L_8} to replace the L_8 nonterminal in the structure, and a digit terminal from the list T_{D_3} to replace the D_3 nonterminal. There are 7 elements in T_{L_8} and 11 elements in T_{D_3} so there are 77 total guesses that this PCFG can produce. The goal of building a lookup table is to sort these guesses by probability.

Let us begin by recording explicit guesses: for each terminal t_1 in T_{L_8} and for each terminal t_2 in T_{D_3}, we record the guess $t_1||t_2$ where $||$ is the concatenation operator. The left table in Figure 4.4 shows the guesses we get in this manner. Note that these guesses are not sorted. We need to sort them to produce our lookup table, which is the table on the right in Figure 4.4.

Probability	Guess
0.1	password123
0.01	password000
0.01	password111
...	
0.1	baseball123
0.01	baseball000
...	
0.075	princess123
0.0075	princess000
...	

Probability	Guess number	Guess
0.1	1	password123
0.1	2	baseball123
0.1	3	iloveyou123
0.075	4	princess123
	...	
0.01	8	password000
0.01	9	password111
0.01	10	password222
0.01	11	password333
0.01	12	password444
0.01	13	password555
0.01	14	password666
0.01	15	password777
0.01	16	password888
0.01	17	password999
0.01	18	baseball000
	...	

Figure 4.4: Example records produced by iterating over the PCFG of Figure 4.3 to produce explicit guesses. On the left, we show the initial records produced by iterating over the grammar. On the right, we show the lookup table after sorting. Both tables contain 77 rows. For brevity, several entries in each table are skipped.

We can notice immediately that the lookup table on the right has many guesses with the same probability: guesses 1 through 3, for example. Guesses 8 through 37 also all have the same probability, and we know this even though guesses past 18 are not shown. How do we know this? We can inspect the grammar in Figure 4.3 and see that the first three elements in T_{L_8} combined with the last ten elements in T_{D_3} produce thirty guesses. Since the three T_{L_8} terminals have the same probability, and the ten T_{D_3} terminals share a probability, these thirty guesses will also share a probability.

The key insight that makes generating patterns over guesses worthwhile is that we can know this fact *before* recording any guesses, with much less work than would be required to record guesses. Let $|T_{L_8}|$ be the number of terminals in T_{L_8} and let $|T_{D_3}|$ be defined the same way. The number of explicit guesses we record is $|T_{L_8}| \times |T_{D_3}| = 77$. However, the total number of terminals we need to inspect to find groups of terminals that share a probability is $|T_{L_8}| + |T_{D_3}| = 18$. We need to sort the terminal lists to find terminal groups, but again this is much less work than sorting the list of explicit guesses.

We call a group of terminals that have the same probability and are produced by the same nonterminal a *terminal group*. The five terminal groups in the grammar

Probability	Size of pattern	Pattern
0.1	3	$L_8D_3\langle 0,0\rangle$
0.01	30	$L_8D_3\langle 0,1\rangle$
0.075	2	$L_8D_3\langle 1,0\rangle$
0.0075	20	$L_8D_3\langle 1,1\rangle$
0.05	2	$L_8D_3\langle 2,0\rangle$
0.005	20	$L_8D_3\langle 2,1\rangle$

Probability	Guess number	Pattern
0.1	1	$L_8D_3\langle 0,0\rangle$
0.075	4	$L_8D_3\langle 1,0\rangle$
0.05	6	$L_8D_3\langle 2,0\rangle$
0.01	8	$L_8D_3\langle 0,1\rangle$
0.0075	38	$L_8D_3\langle 1,1\rangle$
0.005	58	$L_8D_3\langle 2,1\rangle$

Figure 4.5: Example records produced by iterating over terminal groups in a PCFG. See Figure 4.3 to find the relevant terminal groups. In contrast with Figure 4.4, which has 77 rows in each table, recording patterns allows us to only need 6, yet we cover all 77 guesses. On the left, we show the initial records produced by iterating over the terminal groups. On the right, we show the lookup table after sorting. We identify each pattern by the structure and ranks of the terminal groups that produced it. The terminal-group ranks are shown in angle brackets, ordered from left to right based on the nonterminals in the structure. They are zero-based so that the highest-probability terminal group in a list has rank 0. The size of a pattern is the number of guesses it produces. The probability of a pattern is not the cumulative probability of all of its guesses, but the probability of an individual guess. This is required so that our lookup table represents the correct ordering of explicit guesses.

are identified in Figure 4.3 by square brackets. There are three terminal groups in T_{L_8} and two terminal groups in T_{D_3}.

Knowing where the terminal groups are, we can iterate over terminal groups instead of terminals, and produce patterns instead of explicit guesses. We show this with our example PCFG in Figure 4.5. This requires only six records to be produced and sorted, as opposed to 77. Instead of storing an explicit guess in each record, we store an identifier for the pattern that points at a specific structure and terminal groups.

Surprisingly, we have lost no information by doing this. Every explicit guess in Figure 4.4 is covered by a pattern in Figure 4.5. Further, no guesses are placed out of order. The order of the sorted patterns in Figure 4.5 matches the ordering of explicit guesses in Figure 4.4.[13]

We can also produce all guesses from a given pattern easily: simply iterate over all terminals within each terminal group and concatenate the terminals together.

[13]It is possible for two guesses to have the same probability yet be produced by different structures, so it is possible for the guesses of one pattern to be interrupted by the guesses of another pattern, depending on the order in which they are generated. However, we assume that the ordering of guesses beyond probability does not matter, so we can choose to put all of the guesses from a single pattern together in a contiguous block.

Thus, if needed, we could expand the lookup table of patterns into a lookup table of explicit guesses. In Section 4.7, we show how we can look up a password without having to expand the table.

Increasing advantage

The advantage of patterns over explicit guesses increases as the size of a structure increases. For example, a pattern with two nonterminals covers a number of guesses that is a product of two numbers. This was the case with the PCFG in Figure 4.3. If a pattern contains five nonterminals, it covers a number of guesses that is the product of five numbers. Thus, patterns allow us to represent thousands, or potentially far more, guesses with no reduction in the accuracy of our model. In the worst case, all terminals have distinct probabilities, and patterns provide no gain in efficiency, but this is not much worse than generating guesses explicitly. In Section 5.2, we introduce the concept of quantization, which replaces the probabilities of a group of terminals with their average probability. This trades accuracy for space, allowing us to artificially increase the advantage of patterns over explicit guesses.

4.4 Intelligent skipping

In Section 4.3, we explained that it is not feasible to iterate over the space of strings in a Weir PCFG, because the space is too large. So, we require that the user choose a *probability cutoff*. In this section, we explain how one can use a probability cutoff to iterate over a Weir PCFG in an efficient manner. Such an algorithm is needed because the space of patterns in a PCFG can also be very large. We find that a PCFG trained for a `basic8` policy[14] produces over 10^{22} patterns. Naïvely iterating over this space to find all patterns above the cutoff is not practical.

We call this algorithm *intelligent skipping*. To illustrate the algorithm, we use the example grammar of Figure 4.3 from Section 4.3. It is assumed that the reader has read Section 4.3 and understands the concept of terminal groups.[15] The inputs to the algorithm are a PCFG, a particular structure from that PCFG, and a probability cutoff. The algorithm identifies all patterns descended from the input structure that are above or equal to the given cutoff.

[14] Our convention for naming policies is described in Section 2.4.4.

[15] Even though we present and implement our algorithm using terminal groups, it could be applied to the explicit terminals of a grammar as well by simply forcing each terminal into its own group.

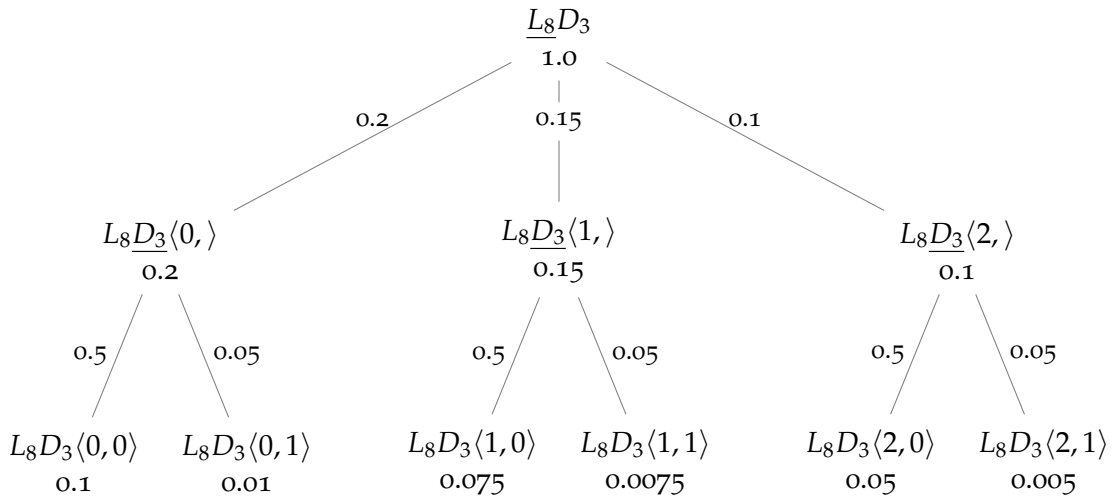


Figure 4.6: Sorted structure-tree for the PCFG of Figure 4.3. An edge from parent to child indicates that a nonterminal in the parent has been substituted in the child for a terminal group. Each node contains two elements: a partial pattern and a probability value (see text for more details). The nonterminal to be substituted is underlined in each internal node. Labels on edges indicate probabilities of substitution and come from the right-hand-side values in Figure 4.3. Note that the leaf probabilities match the probabilities in the left-most column of Figure 4.5.

The actual algorithm used by the guess-calculator framework is presented in Section 4.4.4, but its operation is unintuitive. To make its operation more clear, we first present a similar, parallel algorithm that uses a special tree structure and modified depth-first traversal. After the fundamental concepts of this tree-based algorithm have been explored, we present our final algorithm and show how it maps back to the tree.

The tree-based algorithm is presented starting in the next section.

4.4.1 Sorted structure-trees

First, we introduce the data structure that underlies our algorithm. We call it a *sorted structure-tree*. It is a tree with a variable number of children per node, where each node contains two elements: a *partial pattern* and a probability value. A partial pattern is formed by taking a structure and substituting zero or more nonterminals in it with corresponding terminal groups. Examples of partial patterns include: L_8D_3; $L_8D_3\langle 0,\rangle$ which represents the L_8D_3 structure with the highest probability terminal group of L_8 substituted in; and $L_8D_3\langle 0,1\rangle$ where both nonterminals have been substituted. Both structures and patterns are instances of partial patterns. We use the shorthand `Pr[n]` to represent the probability of a node `n`.

Construction

We can use the following algorithm to construct an SST.

Algorithm C (*Construct an SST*). Given a Weir PCFG `G` and a structure `S`, construct a tree by expanding the leftmost nonterminal in each node until there are no nonterminals left to expand.

Let `n` be the number of nonterminals in the input structure and let `N[0]`, ..., `N[n-1]` index those nonterminals from left to right.

C1. [Create the root.] The root node contains `S` and the probability $\theta_{\mathcal{S}\to \mathtt{S}}$ from `G`.

C2. [Initialize at root.] Set depth $\mathtt{d} \leftarrow 0$.

C3. [Expand nonterminal in each leaf.] Iterate over the terminal groups in `N[d]` in decreasing probability order. Starting with the highest probability group, for each terminal group `T` in `N[d]` and each node `x` at depth `d`, add child `c` to `x` with the following values.
The weight of the edge from `x` to `c` is the probability of `T`.[16]
The partial pattern in `c` is the partial pattern from `x` with `N[d]` replaced with `T`.
The probability of `c` is given by $\mathtt{Pr[c]} = \mathtt{Pr[x]} \cdot \mathtt{Pr[T]}$.
(Let $|\mathtt{T}|$ be the number of terminal groups in `N[d]`. If there are `k` leaves in the tree before this step, there will be $|\mathtt{T}| \cdot \mathtt{k}$ new leaves created.)

C4. [Go down a level.] If $\mathtt{d} < \mathtt{n} - 1$, set $\mathtt{d} \leftarrow \mathtt{d} + 1$ and return to step C3. ▮

The SST produced from our example grammar of Figure 4.3 is given in Figure 4.6.

Properties

All SSTs must satisfy the following properties:

1. The probability of each node is equal to the product of the root-node probability and the probabilities along the edges on the path from the root.
2. All edge weights are in $(0, 1]$.
3. Sibling nodes are sorted from left to right in order of decreasing probability. In other words, all right siblings of a node `n` have a probability less than or equal to `Pr[n]`.
4. At a given tree depth `d`, where `d` is less than the tree height `h`, all nodes substitute the same nonterminal to produce children. For example, the root

[16] Remember that all terminals within a terminal group have the same probability, so we define `Pr[T]` as the probability of any terminal in `T`: $\theta_{\mathtt{N[d]}\to\mathtt{t}}$ where $\mathtt{t} \in \mathtt{T}$.

node in Figure 4.6 produces children by substituting the first nonterminal in the structure, L_8. All nodes at depth 1 in Figure 4.6 (the middle layer) replace the second nonterminal to produce their children. The bottom layer of leaf nodes have no nonterminals to substitute.

5. The SST is *complete*. The only leaves are at the bottom layer of the tree and this layer is completely filled.

These properties follow logically from how the SST was constructed in Algorithm C. You can verify them in Figure 4.6. We will cite these properties when proving the correctness of our algorithm.

4.4.2 Naïve algorithm

Now that we have introduced the sorted structure-tree, we can present a naïve algorithm for traversing it to find patterns above or equal to our probability cutoff. We will improve upon this algorithm in Section 4.4.3.

Our naïve algorithm simply performs a preorder traversal of the SST with the following modification. If a node's probability is below the cutoff, we do not need to explore its children. Since we know from SST properties 1 and 2 that the probability of a node can never be greater than any of its ancestors, we do not need to search further down the tree.

Algorithm P (*Preorder traversal with early return*). Given a root node `n` of an SST and a probability cutoff `cut`, record all patterns with probability above or equal to `cut`.

This algorithm is recursive. If a `n`'s probability is below the cutoff, return. Otherwise, call this algorithm recursively on the children of `n` with the same cutoff.

P1. [Base case: Early return.] If `Pr[n]` $<$ `cut`, return.

P2. [Base case: Leaf node.] If `n` is a leaf node, record `n`. (If `n` has no children then it has no nonterminals to substitute. In other words, it contains a pattern. Since we did not return in P1, our probability is above or equal to `cut` and we should record this pattern.)

P3. [Recurse on children.] Call Algorithm P with `cut` for each child of `n`. ∎

Figure 4.7 shows the result of applying Algorithm P to our example SST with `cut` equal to 0.01. The algorithm behaves as expected and two patterns are not recorded. Figure 4.8 shows what happens with `cut` equal to 0.08. Even though the algorithm has the ability to return without visiting child nodes, it is forced to visit all nodes in this case. Since none of the internal nodes have a probability less than the cutoff, the early return is not triggered.

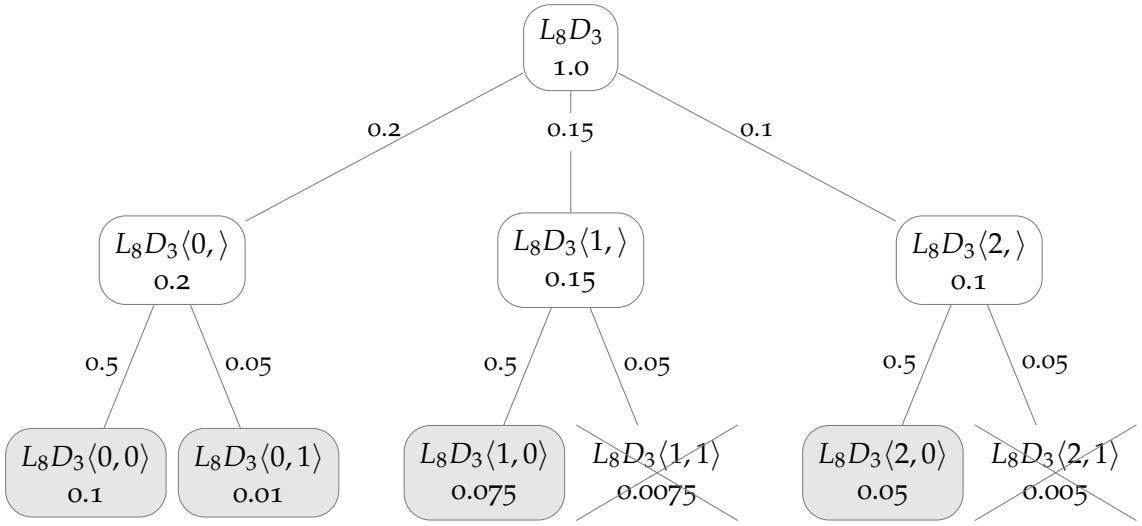


Figure 4.7: SST after applying Algorithm P with a probability cutoff of 0.01. Visited nodes are outlined, nodes whose patterns were recorded are shaded, and nodes whose patterns were rejected (below the cutoff) are crossed out. Two patterns were not recorded.

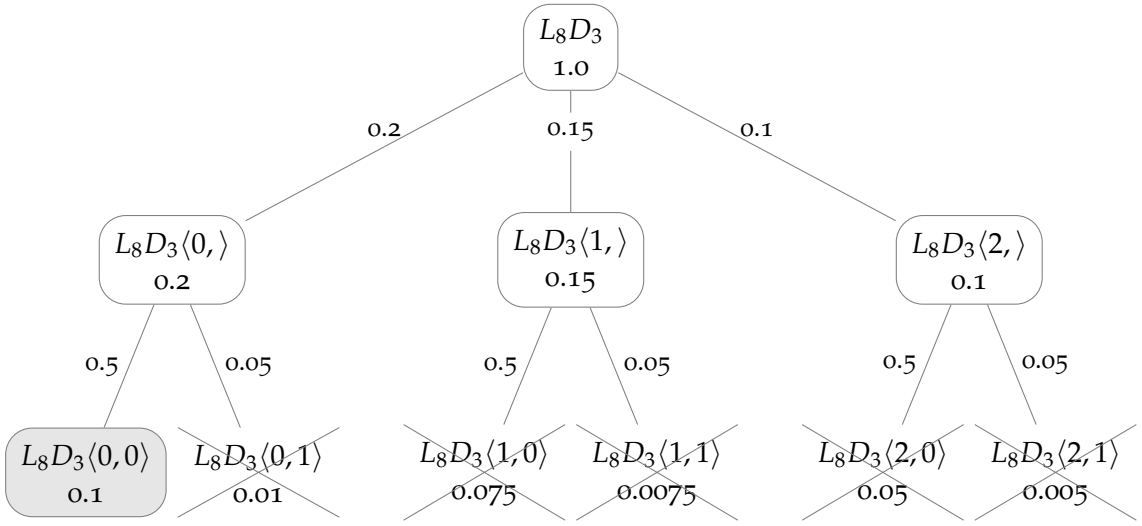


Figure 4.8: SST after applying Algorithm P with a probability cutoff of 0.08. Visited nodes are outlined, nodes whose patterns were recorded are shaded, and nodes whose patterns were rejected (below the cutoff) are crossed out. Only one pattern was recorded, but all nodes were visited because no internal node had a probability less than the cutoff.

4.4.3 Intelligent algorithm

We can observe that Algorithm P has an obvious inefficiency. SST property 3, which requires that all siblings are sorted, implies that once a node is rejected, we can reject all of its siblings on the right without even visiting them. This is the first improvement to our naïve algorithm. Surprisingly, SST property 4 allows us to go even further. If the node we reject happens to be the leftmost (highest probability) child, we can reject all of its siblings **and** all of its parent's right siblings! We call this *intelligent skipping*. Later in this section, we provide a proof that this behavior is safe.

First we present our algorithm, then we show how it applies to our example SST.

Algorithm I (*Preorder traversal with intelligent skipping*). Given a root node `n` of an SST and a probability cutoff `cut`, record all patterns with probability above or equal to `cut`. Like Algorithm P, this algorithm is recursive. Unlike Algorithm P, however, the algorithm returns a Boolean value. This value indicates to a node's parent whether or not to continue.

I1. [Base case: Early return.] If `Pr[n]` $<$ `cut`, return `FALSE`. (This node is below the cutoff, so its parent should ignore its remaining siblings.)

I2. [Base case: Leaf node.] If `n` is a leaf node, record `n` and return `TRUE`. (Record this pattern and return, telling parent to continue.)

I3. [Initialize pointer to first child.] Set `c` to be the leftmost child of `n`. Set `fc` $\leftarrow$ `TRUE`. (We start with the highest probability child of `n`, and set the first-child flag `fc` to `TRUE`.)

I4. [Recurse on child.] Set `r` to the return value of Algorithm I called with `c` and `cut`.

I5. [Continue?] If `r` $=$ `TRUE`, go to step I6. Otherwise, go to step I7. (If `c`'s probability is above or equal to the cutoff, we can move to the next child and continue. Otherwise, we terminate.)

I6. [Update pointer and repeat.] If `c` is the rightmost child of `n`, return `TRUE`. Otherwise, set `c` to be the immediate right sibling of `n`, set `fc` $\leftarrow$ `FALSE`, and go to step I4. (If we reached here and `n` has no more children, return `TRUE`. Otherwise, recurse on the next child.)

I7. [First child?] If `fc` $=$ `TRUE`, return `FALSE`. Otherwise return `TRUE`. (In either case we terminate, since we encountered a child below the cutoff. However, if the first child is below the cutoff, we return `FALSE` to indicate to our parent that all of our children were skipped, and all of our parent's right siblings can be

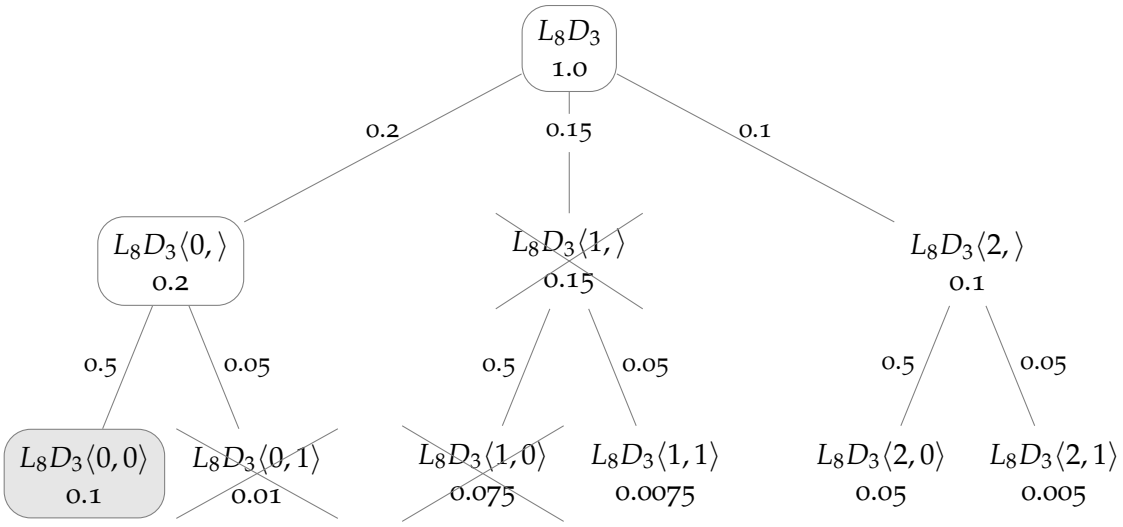


Figure 4.9: SST after applying Algorithm I with a probability cutoff of 0.08. Visited nodes are outlined, nodes whose patterns were recorded are shaded, and nodes whose patterns were rejected (below the cutoff) are crossed out. Once a node is rejected, we do not visit its right siblings. If the node is a leftmost child, we reject the parent and do not visit its parent's right siblings. The middle node in the middle layer of the tree was first visited and then later rejected when its leftmost child was rejected. Compare with Figure 4.8—the outcome is the same, that only one pattern is recorded, but we are able to *skip* four nodes.

skipped. Otherwise, we skip all remaining children but our parent proceeds as usual.) ▮

Figure 4.9 shows the result of applying Algorithm I to our example SST with `cut` equal to 0.08. We are now able to skip four of the nodes and terminate before even visiting the rightmost branch of the tree.

Though this example is simple, intelligent skipping allows us to iterate over a huge space of patterns in a reasonable amount of time. For large trees with many layers, we are able to skip almost all nodes, since many nodes are below the cutoff.

Correctness

Intelligent skipping differs from Algorithm P in two ways. First, we skip all right siblings of a node below the cutoff. This follows logically from SST property 3. Second, if a leftmost sibling is below the cutoff, we skip all of its parent's right siblings. The correctness of this second property is not obvious, so we provide an informal proof of it here.

Theorem I. Given an SST, if a node's probability is below the cutoff, and it is a leftmost child, all children of its parent's right siblings are below the cutoff.

Proof. By SST property 4, all nodes at a given depth expand the same nonterminal. Therefore, the edge weight from a node to its leftmost child will be the same weight as on the edge from any right sibling to that right sibling's leftmost child. Let us call the node $\mathtt{n}$, its leftmost child $\mathtt{n}_\mathtt{L}$, a right sibling $\mathtt{r}$, and that right sibling's leftmost child $\mathtt{r}_\mathtt{L}$. SST property 4 ensures that:

$$\mathtt{Pr}[\mathtt{n} \to \mathtt{n}_\mathtt{L}] = \mathtt{Pr}[\mathtt{r} \to \mathtt{r}_\mathtt{L}] \tag{4.4.1}$$

By SST property 3, we know that $\mathtt{Pr}[\mathtt{n}] \geq \mathtt{Pr}[\mathtt{r}]$. In other words, the right sibling has probability less than or equal to our node, since siblings are sorted. Thus, we can derive the following:

$$\begin{aligned}
\mathtt{Pr}[\mathtt{n}] &\geq \mathtt{Pr}[\mathtt{r}] && \\
\mathtt{Pr}[\mathtt{n} \to \mathtt{n}_\mathtt{L}] \cdot \mathtt{Pr}[\mathtt{n}] &\geq \mathtt{Pr}[\mathtt{r}] \cdot \mathtt{Pr}[\mathtt{n} \to \mathtt{n}_\mathtt{L}] && \text{by SST property 2} \\
\mathtt{Pr}[\mathtt{n} \to \mathtt{n}_\mathtt{L}] \cdot \mathtt{Pr}[\mathtt{n}] &\geq \mathtt{Pr}[\mathtt{r}] \cdot \mathtt{Pr}[\mathtt{r} \to \mathtt{r}_\mathtt{L}] && \text{by (4.4.1)} \\
\mathtt{Pr}[\mathtt{n}_\mathtt{L}] &\geq \mathtt{Pr}[\mathtt{r}_\mathtt{L}] && \text{by SST property 1}
\end{aligned}$$

Therefore if we reject $\mathtt{n}_\mathtt{L}$, we should also reject $\mathtt{r}_\mathtt{L}$, the right sibling's leftmost child, since $\mathtt{Pr}[\mathtt{n}_\mathtt{L}] \geq \mathtt{Pr}[\mathtt{r}_\mathtt{L}]$. By SST property 3, we can also reject all siblings of $\mathtt{n}_\mathtt{L}$ and $\mathtt{r}_\mathtt{L}$. Finally, since we did not specify a particular right sibling for $\mathtt{r}$, the proof applies to all right siblings of $\mathtt{n}$. ∎

4.4.4 Using mixed-radix numbers

Instead of using a tree to represent all of the possible patterns produced by a structure, we use a mixed-radix number.[17] In fact, you have already seen this number in the form $\langle 2, 0 \rangle$ in Figure 4.5 and the various SST diagrams, such as Figure 4.9.

We describe how this number works with an example, which you can follow on Figure 4.9. Take the mixed-radix number $\langle 2, 0 \rangle$. To reach the node with this index, start at the root, and move to the third child from the left (which happens to be the rightmost child), and then to the leftmost child of that node. This places us at the leaf with index $\langle 2, 0 \rangle$. To clarify, we took the third-highest-probability terminal group for the first nonterminal in the structure, and then the highest probability terminal group for the second nonterminal. Since the mixed-radix number is zero-indexed, a 0 corresponds to the leftmost child on an SST. Remember that

[17] `https://en.wikipedia.org/wiki/Mixed_radix`

the leftmost child is special in our intelligent skipping algorithm, so a 0 in our mixed-radix number is special in the same way.

A mixed-radix number is defined by two quantities: the number itself and the radices that correspond to each position in the number. For example, the number $\langle 2, 0\rangle$ has radices $[3, 2]$; the root node has three children, and each node at depth 1 has two children. Each position in the mixed-radix number corresponds to a nonterminal in the structure, and a branch taken in the SST. The mixed-radix number serves to index the leaves in the SST, and the number of possible values of the number is equal to the number of leaves in the tree.

We can increment the mixed-radix number to move to the next leaf in the tree that we would reach using preorder traversal. For example, incrementing $\langle 2, 0\rangle$ gives $\langle 2, 1\rangle$ which is the immediate right sibling of $\langle 2, 0\rangle$. Incrementing $\langle 1, 1\rangle$ gives $\langle 2, 0\rangle$, the leftmost child of $\langle 1, 1\rangle$'s parent's right sibling. Algorithms for incrementing mixed-radix numbers are well known and easy to implement.[18]

The mixed-radix number does not model any of the internal nodes of the tree. Instead, our algorithm skips directly from a leaf below the cutoff to the next leaf that we need to evaluate. We present the algorithm and then some short examples to provide an intuition into how it works.

Algorithm M (*Intelligent skipping with a mixed-radix number*). Given a Weir PCFG `G`, a structure `S`, and a probability cutoff `cut`, print all patterns with probability above or equal to `cut`.

Let `n` be the number of nonterminals in `S` and let `N[0]`, ..., `N[n-1]` index those nonterminals from left to right. Let $\texttt{p}_\texttt{r} =$`[|N[0]|, ..., |N[n-1]|]`, where `|N[i]|` is the number of terminal groups produced by `N[i]`. (These are the radices of our mixed-radix number.)

M1. [Initialize mixed-radix number.] Set `p` $\leftarrow \langle 0, \ldots, 0\rangle$, such that `p` has `n` digits. (This step initializes our mixed-radix number to the highest probability pattern that can be produced by the structure. In SST terms, 0 corresponds to the leftmost child of a node, so `p` points at the leftmost child of the leftmost child, etc. down the tree.)

M2. [Check probability.] Compute:

$$\texttt{Pr[p]} = \texttt{Pr[S]} \cdot \prod_{i=0}^{\texttt{n}-1} \texttt{Pr}\big[\texttt{N[i][p[i]]}\big]$$

where `N[i][j]` is the $\texttt{j}^{th}$ terminal group of the nonterminal `N[i]`.

If `Pr[p]` is above or equal to `cut`, go to step M3. Otherwise, go to step M5.

[18] One such algorithm is Knuth's Algorithm 7.2M [74].

M3. [Record pattern.] Record `p`.

M4. [Move to next pattern.] Increment `p` and go to step M2. If `p` cannot be incremented, terminate. (Move to the next leaf and check its probability. If there are no more leaves, then we are done traversing the structure.)

M5. [Skip ahead.] Scan `p` from right to left for the first nonzero value. Set this digit to its maximum value and do the same for all digits to its right. Then go to step M4. If `p` has no nonzero values, terminate.
(Let `i` be the position of the first nonzero value in `p`. `p[i]`'s maximum value is $\texttt{p}_\texttt{r}\texttt{[i]} - 1$. We set $\texttt{p[i]} \leftarrow (\texttt{p}_\texttt{r}\texttt{[i]} - 1)$, $\texttt{p[i+1]} \leftarrow (\texttt{p}_\texttt{r}\texttt{[i+1]} - 1)$, ..., $\texttt{p[n-1]} \leftarrow (\texttt{p}_\texttt{r}\texttt{[n-1]} - 1)$. This sets `p` to the last leaf whose probability is known to be below `cut`. We then go to step M4 to increment `p` and continue. A more detailed explanation of what is happening in this step is provided below.) ▌

Intelligent skipping is implemented in step M5. A few examples will help to clarify how this step works. Take $\texttt{p} = \langle 0, 1\rangle$ with radices $[3, 2]$ as in Figure 4.9. Scan `p` from right to left for the first nonzero value; it is found immediately in the rightmost digit: 1. Step M5 sets this digit to its maximum value, which is also 1. Therefore, we have not changed the value of `p`.

Now imagine that the radices of `p` are $[3, 5]$. In other words, $\langle 0, 1\rangle$ has a left sibling: $\langle 0, 0\rangle$, and three right siblings: $\langle 0, 2\rangle$, $\langle 0, 3\rangle$, and $\langle 0, 4\rangle$.[19] Step M5 sets `p` to $\langle 0, 4\rangle$, the rightmost sibling. We will next go to step M4 and increment `p`, which sets `p` to $\langle 1, 0\rangle$. Thus, we skip all the right siblings of $\langle 0, 1\rangle$. This is the same approach Algorithm I would have taken.

Since we were not at a leftmost child, we did not skip any higher set of siblings. Next, we see an example of that behavior. Take $\texttt{p} = \langle 0, 0, 1, 0\rangle$ with radices $[3, 3, 5, 5]$. The first nonzero value is found in the second digit from the right. Step M5 sets `p` to $\langle 0, 0, 4, 4\rangle$. Since we are at a leftmost child, identified by a 0 in the last digit, we want to skip all of our *parent*'s right siblings. Therefore, we set the third digit, which represents the second layer from the bottom of the SST, to its maximum value. After incrementing, we have $\texttt{p} = \langle 0, 1, 0, 0\rangle$.[20]

Conclusion

Our algorithm maps a mixed-radix number onto the leaves of an SST, indexing all patterns for a given structure. It starts with $\langle 0, \ldots, 0\rangle$, which is the highest

[19] Try imagining each node in the middle layer of Figure 4.9 with five children. We are currently on the second one.

[20] Note that we are now set up nicely in case `Pr[p]` is below the cutoff, since we are at the leftmost child of a leftmost child. If we need to skip again from here, we would be at $\texttt{p} = \langle 1, 0, 0, 0\rangle$.

probability pattern that a structure can produce. When the algorithm encounters a pattern below a given probability cutoff, it skips ahead along the tree, past patterns whose probability can be logically deduced to be below the cutoff. It is possible that the algorithm can be optimized further, but we have found that it is able to record patterns from an enormous possible space in a reasonable amount of time.

In Section 4.7, we will explain how our lookup process acts as a check on Algorithm M. If a password in the test set has a probability above the cutoff and a pattern that was not recorded, the lookup process returns an error and flags this password. We have been using Algorithm M as part of the guess-calculator framework for a few years. In that time, we have looked up several thousand passwords without seeing an error. We believe that, given the same inputs, Algorithm M will record the same patterns as Algorithm I composed with Algorithm C, though its correctness is not intuitive. It is also much easier to implement.

4.5 Pattern compaction

In this section, we introduce an optimization to the way we record patterns. We call it *pattern compaction*. We discovered the need for it after examining the entries in our lookup table and noticing that many of them share the same probability and the same structure. For some structures, pattern compaction allows some patterns descended from this structure to be collapsed into a single pattern in a way that is deterministic and still allows these patterns to be looked up later. First, we present a PCFG where this optimization can be applied, then we explain how the algorithm works.

The PCFG shown in Figure 4.10, which has a modified structure compared to our previous example PCFG in Figure 4.3, produces patterns that can be compacted. The structure $D_3L_8D_3$ has the D_3 nonterminal repeated, once at the beginning of the password and once at the end. This will produce passwords like `123password000`. The patterns produced by this PCFG are shown in Figure 4.11.

The pattern that covers `123password000` is $D_3L_8D_3\langle 0,0,1\rangle$ using our mixed-radix notation. As you can see from Figure 4.11, this pattern covers 30 guesses and its probability is 0.005. Let us say that our probability cutoff is below or equal to this number. We record the pattern $D_3L_8D_3\langle 0,0,1\rangle$. Our key insight is that, by the definition of a *context-free* grammar, the password `123password000` has the same probability as `000password123`. The ordering of the terminals in

Θ:

Structures

$\theta_{\mathcal{S}\to D_3L_8D_3} = 1.0$

Alphabetic terminals (T_{L_8})		Digit terminals (T_{D_3})	
$\theta_{L_8\to\text{password}}$	$= 0.2$	$\theta_{D_3\to 123}$	$= 0.5$
$\theta_{L_8\to\text{baseball}}$	$= 0.2$	$\theta_{D_3\to 000}$	$= 0.05$
$\theta_{L_8\to\text{iloveyou}}$	$= 0.2$	$\theta_{D_3\to 111}$	$= 0.05$
$\theta_{L_8\to\text{princess}}$	$= 0.15$	$\theta_{D_3\to 222}$	$= 0.05$
$\theta_{L_8\to\text{sunshine}}$	$= 0.15$	$\theta_{D_3\to 333}$	$= 0.05$
$\theta_{L_8\to\text{superman}}$	$= 0.1$	$\theta_{D_3\to 444}$	$= 0.05$
$\theta_{L_8\to\text{whatever}}$	$= 0.1$	$\theta_{D_3\to 555}$	$= 0.05$
		$\theta_{D_3\to 666}$	$= 0.05$
		$\theta_{D_3\to 777}$	$= 0.05$
		$\theta_{D_3\to 888}$	$= 0.05$
		$\theta_{D_3\to 999}$	$= 0.05$

Figure 4.10: Weir PCFG example with a repeated nonterminal (D_3). The repeated nonterminal causes the patterns produced by this structure to have repeated probabilities as shown in Figure 4.11.

a password does not alter its probability.[21] Of course, the model is simpler than reality, where the passwords `123password000` and `000password123` probably have different probabilities. Of the models we have discussed thus far, only the Simpleguess model from Section 3.1 can capture such subtleties. Our Weir PCFGs do not.[22]

The passwords `123password000` and `000password123` have two different mixed-radix numbers associated with them: $D_3L_8D_3\langle 0,0,1\rangle$ and $D_3L_8D_3\langle 1,0,0\rangle$. We use the notation $L_8\langle 0\rangle$ to identify the first terminal group produced by the nonterminal

[21] Our model captures some orderings, but not others. The string `password123` will probably have a different probability than `123password` because the two strings have different structures. The former password has structure L_8D_3 and the latter D_3L_8. Digits are more likely at the end of a password than in the middle, so L_8D_3 probably has a higher probability than D_3L_8. Only reorderings of terminals that *do not* change the structure will share the same probability.

[22] Hybrid structures, which are introduced in Section 5.5.1, will be able to capture this as well.

Probability	Size of pattern	Pattern
0.05	3	$D_3L_8D_3\langle 0,0,0\rangle$
0.005	30	$D_3L_8D_3\langle 0,0,1\rangle$
0.0375	2	$D_3L_8D_3\langle 0,1,0\rangle$
0.00375	20	$D_3L_8D_3\langle 0,1,1\rangle$
0.025	2	$D_3L_8D_3\langle 0,2,0\rangle$
0.0025	20	$D_3L_8D_3\langle 0,2,1\rangle$
0.005	30	$D_3L_8D_3\langle 1,0,0\rangle$
0.0005	300	$D_3L_8D_3\langle 1,0,1\rangle$
0.00375	20	$D_3L_8D_3\langle 1,1,0\rangle$
0.000375	200	$D_3L_8D_3\langle 1,1,1\rangle$
0.0025	20	$D_3L_8D_3\langle 1,2,0\rangle$
0.00025	200	$D_3L_8D_3\langle 1,2,1\rangle$

Probability	Guess number	Pattern
0.05	1	$D_3L_8D_3\langle 0,0,0\rangle$
0.0375	4	$D_3L_8D_3\langle 0,1,0\rangle$
0.025	6	$D_3L_8D_3\langle 0,2,0\rangle$
0.005	8	$D_3L_8D_3\langle 0,0,1\rangle$
0.005	38	$D_3L_8D_3\langle 1,0,0\rangle$
0.00375	68	$D_3L_8D_3\langle 0,1,1\rangle$
0.00375	88	$D_3L_8D_3\langle 1,1,0\rangle$
0.0025	108	$D_3L_8D_3\langle 0,2,1\rangle$
0.0025	128	$D_3L_8D_3\langle 1,2,0\rangle$
0.0005	148	$D_3L_8D_3\langle 1,0,1\rangle$
0.000375	448	$D_3L_8D_3\langle 1,1,1\rangle$
0.00025	648	$D_3L_8D_3\langle 1,2,1\rangle$

Figure 4.11: Pattern records produced by iterating over the PCFG of Figure 4.10. On the left, we show the records produced by Algorithm 4.4M with no probability cutoff. Notice that the mixed-radix number associated with each pattern is incremented in each entry. On the right, we show the lookup table after sorting. The lookup table has 12 entries and covers 847 guesses. Notice that lookup-table entries 4 and 5 have the same probability, as well as 6 and 7, and 8 and 9. We can know that this repetition will occur *a priori*, and compact these repetitions into single patterns. This is shown in Figure 4.12.

L_8. The probability of $D_3L_8D_3\langle 0,0,1\rangle$ is:

$$\texttt{Pr}[D_3L_8D_3] \cdot \texttt{Pr}[D_3\langle 0\rangle] \cdot \texttt{Pr}[L_8\langle 0\rangle] \cdot \texttt{Pr}[D_3\langle 1\rangle]$$

and for $D_3L_8D_3\langle 1,0,0\rangle$:

$$\texttt{Pr}[D_3L_8D_3] \cdot \texttt{Pr}[D_3\langle 1\rangle] \cdot \texttt{Pr}[L_8\langle 0\rangle] \cdot \texttt{Pr}[D_3\langle 0\rangle]$$

Therefore, both patterns have the same probability.

When a structure contains a repeated nonterminal, and that nonterminal has at least two terminal groups, we can anticipate patterns that have the same probability. Notice that lookup-table entries 4 and 5 in Figure 4.11 have the same probability, as well as 6 and 7, and 8 and 9. We can identify such entries by looking for permutations of terminal-group indices under repeated nonterminals. Consider the partial pattern $D_3L_8D_3\langle 0,-,0\rangle$, where we ignore the second digit since the L_8 nonterminal is not repeated. This pattern has only a single permutation, because exchanging the 0s does not produce a new number. However, the pattern $D_3L_8D_3\langle 0,-,1\rangle$ has two permutations: $D_3L_8D_3\langle 0,-,1\rangle$ and $D_3L_8D_3\langle 1,-,0\rangle$. Therefore, when we encounter a pattern descended from $D_3L_8D_3\langle 0,-,1\rangle$, such as

entries 4, 6, and 8 from Figure 4.11, we can anticipate the existence of additional patterns with the same probability.

Consider a more complex example: $L_1D_1L_1D_1L_1D_1L_1D_1\langle -,0,-,0,-,0,-,1\rangle$. This partial pattern can produce passwords like `a1b2c3d4`. It has four permutations, because the 1 can be moved around to any of the D_1 spots. We can do even better when the terminal-group indexes do not repeat. For example, $L_1D_1L_1D_1L_1D_1L_1D_1\langle -,0,-,1,-,2,-,3\rangle$ has 24 permutations:

$$L_1D_1L_1D_1L_1D_1L_1D_1\langle -,0,-,1,-,2,-,3\rangle$$
$$L_1D_1L_1D_1L_1D_1L_1D_1\langle -,0,-,1,-,3,-,2\rangle$$
$$L_1D_1L_1D_1L_1D_1L_1D_1\langle -,0,-,2,-,1,-,3\rangle$$
$$L_1D_1L_1D_1L_1D_1L_1D_1\langle -,0,-,2,-,3,-,1\rangle$$
$$\ldots$$

The previous example, has two repeated nonterminals: L_1 and D_1. Therefore, we can also consider the pattern $L_1D_1L_1D_1L_1D_1L_1D_1\langle 0,-,1,-,2,-,3,-\rangle$, where we ignore the indices of the D_1 nonterminal. This can produce up to another 24 permutations:

$$L_1D_1L_1D_1L_1D_1L_1D_1\langle 0,-,1,-,2,-,3,-\rangle$$
$$L_1D_1L_1D_1L_1D_1L_1D_1\langle 0,-,1,-,3,-,2,-\rangle$$
$$L_1D_1L_1D_1L_1D_1L_1D_1\langle 0,-,2,-,1,-,3,-\rangle$$
$$L_1D_1L_1D_1L_1D_1L_1D_1\langle 0,-,2,-,3,-,1,-\rangle$$
$$\ldots$$

Therefore, if we encounter the single pattern $L_1D_1L_1D_1L_1D_1L_1D_1\langle 0,0,1,1,2,2,3,3\rangle$, we can anticipate 575 patterns with the same probability.[23] To compute the number of patterns to anticipate, we use a standard formula for the number of permutations of a multiset, computed independently for each repeated nonterminal.[24] This is utilized in the updated version of Algorithm 4.4M for compacted patterns given below.

Algorithm M (*Intelligent skipping with pattern compaction*). Given a Weir PCFG `G`, a structure `S`, and a probability cutoff `cut`, print all compacted patterns with probability above or equal to `cut`.

Let `n` be the number of nonterminals in `S` and let `N[0]`, ..., `N[n-1]` index those nonterminals from left to right. Let $\mathtt{p_r} =$`[|N[0]|, ..., |N[n-1]|]`, where `|N[i]|` is the number of terminal groups produced by `N[i]`.

[23] $24 \cdot 24 = 576$. After seeing the first one, we expect 575 more.

[24] This formula is given in Appendix A.1.

M1. [Initialize mixed-radix number.] Set $\texttt{p} \leftarrow \langle 0, \ldots, 0 \rangle$, such that p has n digits.

M2. [Check probability.] Compute:

$$\texttt{Pr[p]} = \texttt{Pr[S]} \cdot \prod_{i=0}^{\texttt{n}-1} \texttt{Pr}\big[\texttt{N[i][p[i]]}\big]$$

where N[i][j] is the $\texttt{j}^{th}$ terminal group of the nonterminal N[i].
If Pr[p] is above or equal to cut, go to step M3. Otherwise, go to step M9.

M3. [Can p be compacted?] For each repeated nonterminal in S, check if its corresponding positions in p contain more than one value. If so for any repeated nonterminal, go to step M4. Otherwise, go to step M7.
(If there are no repeated nonterminals in p, do not worry about pattern compaction and go to step M7. If there are repeated nonterminals, we still have to check for the possibility of compaction. For example, a pattern that matches $D_3 L_8 D_3 \langle 0, -, 0 \rangle$ cannot be compacted. Since the positions of p corresponding to the D_3 nonterminal all contain the same value, "0," we cannot anticipate another pattern with the same probability. It has only one permutation. We also record this as a simple pattern by going to step M7.)

M4. [Should p be recorded?] For each repeated nonterminal in S, check if the corresponding digits in p are in nondecreasing order. If so for all repeated nonterminals, go to step M5. Otherwise, go to step M8.
(Compacted patterns reduce the number of records that we produce, but we still visit the same number of leaves. If we encounter a leaf that represents a pattern that has already been recorded, we skip it by going to step M8. For any set of permutations, there is only one permutation where the values are nondecreasing, so this is the one we record.)

M5. [Count permutations.] Set $\texttt{x} \leftarrow 1$. For each repeated nonterminal in S, construct a multiset from the values of the corresponding positions in p. Let y be the number of permutations of this multiset, computed using the standard formula in Appendix A.1. Set $\texttt{x} \leftarrow \texttt{x} \cdot \texttt{y}$ and move to the next repeated nonterminal. (At the end of this step, x contains the total number of permutations of p.)

M6. [Record compacted pattern.] Let m be the original size of p. Record p, but replace its size with $\texttt{m} \cdot \texttt{x}$, where x is the number of permutations from step M5. Go to step M8.

M7. [Record simple pattern.] Record p.

M8. [Move to next pattern.] Increment p and go to step M2. If p cannot be incremented, terminate.

Probability	Size of pattern	Pattern
0.05	3	$D_3L_8D_3\langle 0,0,0\rangle$
0.005	60	$D_3L_8D_3\langle 0,0,1\rangle$
0.0375	2	$D_3L_8D_3\langle 0,1,0\rangle$
0.00375	40	$D_3L_8D_3\langle 0,1,1\rangle$
0.025	2	$D_3L_8D_3\langle 0,2,0\rangle$
0.0025	40	$D_3L_8D_3\langle 0,2,1\rangle$
0.005	-	$D_3L_8D_3\langle 1,0,0\rangle$
0.0005	300	$D_3L_8D_3\langle 1,0,1\rangle$
0.00375	-	$D_3L_8D_3\langle 1,1,0\rangle$
0.000375	200	$D_3L_8D_3\langle 1,1,1\rangle$
0.0025	-	$D_3L_8D_3\langle 1,2,0\rangle$
0.00025	200	$D_3L_8D_3\langle 1,2,1\rangle$

Probability	Guess number	Pattern
0.05	1	$D_3L_8D_3\langle 0,0,0\rangle$
0.0375	4	$D_3L_8D_3\langle 0,1,0\rangle$
0.025	6	$D_3L_8D_3\langle 0,2,0\rangle$
0.005	8	$D_3L_8D_3\langle 0,0,1\rangle$
0.00375	68	$D_3L_8D_3\langle 0,1,1\rangle$
0.0025	108	$D_3L_8D_3\langle 0,2,1\rangle$
0.0005	148	$D_3L_8D_3\langle 1,0,1\rangle$
0.000375	448	$D_3L_8D_3\langle 1,1,1\rangle$
0.00025	648	$D_3L_8D_3\langle 1,2,1\rangle$

Figure 4.12: Compacted patterns produced by iterating over the PCFG of Figure 4.10 using Algorithm 4.5M. Contrast with Figure 4.11 on page 72, which used Algorithm 4.4M. The patterns in gray on the left are still encountered, but we recognize the opportunity for pattern compaction and use it to avoid storing the additional records. The lookup table has 9 entries and still covers 847 guesses.

M9. [Intelligent skip ahead.] Scan p from right to left for the first nonzero value. Set this digit to its maximum value and do the same for all digits to its right. Then go to step M8. If p has no nonzero values, terminate. ∎

We present the result of running our updated Algorithm M in Figure 4.12. Compare the records produced in the left side of Figure 4.11 with those in the left side of Figure 4.12. Those patterns which could be compacted, identified by the partial pattern $D_3L_8D_3\langle 0,-,1\rangle$ had two permutations each, so we multiply their size by 2 when recording. $D_3L_8D_3\langle 0,-,1\rangle$ is an increasing sequence, so those patterns are recorded. We skip the redundant patterns that are encountered later on, identifying that the partial pattern $D_3L_8D_3\langle 1,-,0\rangle$ does not have increasing values.

The takeaway is that taking advantage of repeated nonterminals can greatly reduce the amount of space that the guess-calculator framework needs on disk, with no sacrifice in accuracy. However, it comes with a significant increase in implementation complexity as shown in the previous algorithm and Algorithm 4.7.2C which is needed on lookup. We also recognize that this optimization is extremely specific to our application and might be hard to generalize to other problem domains.

4.6 Parallel computation

A few of the boxes in Figures 4.1 and 4.2 are labeled with "Parallel computation." A major improvement of our approach over Weir's implementation and many other approaches is the use of parallel algorithms. This aspect of our approach increases the usability of the framework, by allowing the entire process to complete in a reasonable amount of time, but does not decrease the total amount of computation performed or the amount of space needed.

In all cases, we use parallelization where the underlying problem is embarrassingly parallel.[25] In this section, we explain where we have employed parallel algorithms.

Quantizing terminal probabilities This improvement will be discussed in more detail in Section 5.2. It is standard practice to run the algorithm that we use [87] a large number of times with randomly selected initial values, and take the best performing iteration. We run these iterations in parallel, each with a different set of initial values. We do this using the `mclapply` command in R.[26]

Generating patterns We parallelize the generation of patterns by running instances of Algorithm 4.5M in parallel, each working on a different input structure from the PCFG. The records produced by each instance are written to separate files.

Sorting We sort these files in parallel by invoking separate instances of the GNU `sort` utility. The `sort` utility has a parallel mode, but can only utilize up to 8 cores at the time of this writing. We use it in non-parallel mode, one process per core. Once this is done, we merge the sorted files into a single sorted file. This merge sort is not parallelized and is a significant bottleneck of the framework.

Assigning guess numbers Each password in the test set can be looked up independently. We divide the test set into `n` chunks, where `n` is the number of cores to parallelize over, and look up these chunks in parallel.

[25] This is a term of art that refers to problems that can be parallelized easily because the individual processes do not need to coordinate. See `https://en.wikipedia.org/wiki/Embarrassingly_parallel`.

[26] `https://stat.ethz.ch/R-manual/R-devel/library/parallel/html/mclapply.html`

4.7 Assigning guess numbers

The output of the guess-calculator framework is the assignment of guess numbers to an input set of test passwords. In this section, we explain how this assignment is performed. Using a PCFG and a lookup table built from that PCFG, we can either assign a guess number to a password, or identify a reason why a guess number cannot be assigned. A guess number cannot be assigned if the PCFG cannot produce the password, or if the lookup table does not extend far enough to assign a guess number.

You have already seen a lookup table in the right side of Figure 4.12 on page 75, and we first described its creation in Section 4.3. The lookup table stores patterns in decreasing probability order, and each record in the lookup table contains three fields: probability, the guess number of the first password in the pattern, and a pattern identifier.

There are many steps involved in assigning a guess number and some of them are complex in and of themselves. We first present our algorithm at a high-level, then we describe some of the steps in more detail. For simplicity, we assume that each password can only be parsed in one way, or not at all. This is true of the Weir PCFGs that we have discussed so far. In Chapter 5, we introduce grammars that can parse a password in multiple ways. The only change needed to accommodate this is to make step A1 below nondeterministic, such that Algorithm A returns a record with the lowest guess number found across all possible parses.

Algorithm A (*Assign guess number to password*). Given a Weir PCFG `G`, a lookup table `L`, and a password `p`, return a record for `p` that includes its guess number, or return a reason why a guess number could not be assigned.

The algorithm makes use of the functions `PARSE` and `PATTERN-OFFSET` that are described later in this chapter. `PARSE` is described in Section 4.7.1, and `PATTERN-OFFSET` is described in Section 4.7.2. It also uses the `SINGLE-RADIX` function presented in Appendix A.3.

A1. [Parse password.] Call `PARSE(p,G)`. If the function is successful, set `pr,pat,t,ids` $\leftarrow$ `PARSE(p,G)`. Otherwise, return "Not in PCFG" and terminate. (If `p` can be produced by `G`, `PARSE` will determine its probability `pr`, pattern identifier `pat`, terminal ranks `t`, and source identifiers `ids`. The pattern identifier looks like $L_8D_3\langle 2,1\rangle$, which encodes both the structure and terminal groups.

A pattern covers many passwords, so we need an additional variable `t` that indexes the given password `p` within `pat`. This is another mixed-radix number provided by the `PARSE` function.

Recall that in Section 4.2 we wrote optional source identifiers into the PCFG for each structure and terminal. These are returned in the `ids` variable.)

A2. [Compute offset from pattern compaction.] Set `fpat, c` $\leftarrow$ `PATTERN-OFFSET(pat)`. (`fpat` is the permutation of `pat` which was recorded in step M6 of Algorithm 4.5M, i.e., for each repeated nonterminal, the corresponding digits in `fpat` are nondecreasing. If `pat` is a compacted pattern, we will not find it in the lookup table. Therefore, we need `fpat` so we can use it as a lookup key.)

A3. [Scan lookup table.] Using `pr` and `fpat` as keys, search `L` for a record that matches both keys. First perform a binary search with `pr` to find the first matching record. Then do a linear scan down the following records until a record matching both `fpat` and `pr` is found. Let `r` be the record found in this way. If `r` exists, go to step A5. Otherwise, go to step A4.
(If `pr` is not found, then the search fails. If the linear scan encounters a record whose probability is not `pr`, then the search also fails, since `L` is sorted by probability.)

A4. [Check for serious error.] If `pr` is within the bounds of `L`, i.e., it is greater than or equal to the probability cutoff used to create `L`, fail the lookup process and return a serious error. Otherwise, return "Password below probability cutoff" and terminate.
(If the probability is above the cutoff, then Algorithm 4.5M should have produced a corresponding pattern. If `fpat` was not found, this indicates an issue with the lookup table creation process and should be taken seriously. During development, this alerted us to issues in the parallel pattern-generation process that we would have otherwise missed. If the probability is below the cutoff, then it will not be found in `L`.)

A5. [Compute guess number.] Let $\mathtt{r}_{\mathtt{GN}}$ be the guess number from the lookup-table record `r`. Set $\mathtt{t}_{\mathtt{GN}} \leftarrow$ `SINGLE-RADIX(t)`, which is the rank of this password within its pattern. Set `c` $\leftarrow 0$ if it is not already set. Let the guess number of the password be $\mathtt{g} \leftarrow \mathtt{r}_{\mathtt{GN}} + \mathtt{t}_{\mathtt{GN}} + \mathtt{c}$.
($\mathtt{r}_{\mathtt{GN}}$ is the guess number found in the lookup table, which is the first password produced by `fpat`. We need to add two offsets:
All permutations of `fpat` are guessed one after another after `fpat`, which is fine because they all share the same probability. The compaction offset, `c`, accounts for `pat` being a compacted pattern that is a later permutation of `fpat`. It tells us how many guesses to skip to get to `pat`, so $\mathtt{r}_{\mathtt{GN}} + \mathtt{c}$ points at the first password produced by `pat`.
$\mathtt{t}_{\mathtt{GN}}$ counts from the start of `pat` to `p`.

Both offsets, c and t_{GN}, can be zero. c is zero if pat is not compacted, or if pat = fpat. t_{GN} is zero if p is the first password produced by pat.)

A6. [Return record.] Return a record including p, g, fpat, pr, and ids. ∎

Algorithm A has four outcomes for each input password:

- The password was parsed successfully and an appropriate pattern was found in the lookup table. We return the guess number of the password.
- The password was parsed and its probability is below the cutoff of the lookup table. We cannot return a guess number in this case, so we record an appropriate code.
- The password was not parsed. This means it cannot be produced by the PCFG, so we do not even need to search the lookup table. We record an appropriate code.
- The password was parsed, its probability is above or equal to the cutoff, and it was not found in the lookup table. This indicates a serious error, and all results should be discarded. This can happen if there was an error during the creation of L, or if the input PCFG G is not the same grammar that was used to generate L.

Recording a guess number or code for all passwords from the test set allows us to look at the output alone and compute percentages of cracked passwords. Our framework is able to take this output and produce guessing curves as introduced in Section 3.2. In previous versions of the framework, we output only the cracked passwords. It quickly became apparent that this makes the analysis more complex.

In the following subsections, we describe the PARSE and PATTERN-OFFSET functions in more detail. Feel free to skip these sections if you are not interested in details of the implementation.

4.7.1 Parsing

Used in step A1 of Algorithm 4.7A, PARSE is a function that takes a password p and a PCFG G, and returns either failure, or a tuple of the password's probability pr, pattern identifier pat, terminal ranks t, and source identifiers ids. In this section, we present an algorithm that implements PARSE.

The simple form of a Weir PCFG makes it easy to parse strings, since the structure of a string can be determined from its length and character-class composition. For example, we know that password123 matches the structure

L_8D_3, though the given PCFG might not contain this structure. It can also match the structure $L_4L_4D_3$, and such structures are introduced in Section 5.5.2. When a password can be parsed in multiple ways, we simply allow `PARSE` to return multiple tuples.

If we can parse `p`, we can assign it a probability under `G`, which is simply the product of the probabilities of all rules used in its production. We can also take the union of all identifiers attached to these rules to get the list of source identifiers (`ids`) to return. The computation of `pat` and `t` are less obvious, but are given by Algorithm P below.

Algorithm P (*Parse password*). Given a Weir PCFG `G` and a password `p`, return failure or a tuple comprised of the password's probability `pr`, pattern identifier `pat`, terminal ranks `t`, and source identifiers `ids`.

We assume that `G` is annotated such that each terminal-producing rule is labeled with the index of the terminal group that it belongs to.

P1. [Initialize structure.] Set `S` to the first structure in `G`. (We will iterate over the structures in `G` one at a time, though we could parallelize here as well.)

P2. [Initialize nonterminals and ids.] Let `n` be the number of nonterminals in `S`, and let `N[0]`, ..., `N[n-1]` index those nonterminals from left to right. Set `ids` to be the value of the identifiers attached to `S` in `G`. If there are no identifiers, set `ids` $\leftarrow \emptyset$.

P3. [Check `p` against `S`.] Compare the length and composition of `p` with `S`. If they match, go to step P5. Otherwise, go to step P4. (If `S` might produce `p`, continue, otherwise, go to the next structure.)

P4. Set `S` to be the next structure in `G` and go to step P2. If there are no more structures in `G`, terminate.

P5. [Tokenize `p`.] Tokenize `p` into $\texttt{p}_0, \ldots, \texttt{p}_{[n-1]}$ such that the length of each $\texttt{p}_i$ matches the length represented by `N[i]`. (Each nonterminal in a Weir PCFG encodes its length, e.g., L_8 represents eight characters. Thus, we can simply tokenize `p` from left to right based on the lengths of each `N[i]`.)

P6. [Initialize cursor and pattern.] Set `i` $\leftarrow 0$. Construct four arrays of size `n`: $\texttt{t}_\texttt{d}$, $\texttt{t}_\texttt{r}$, $\texttt{pat}_\texttt{d}$, and $\texttt{pat}_\texttt{r}$.
(Recall that a mixed-radix number is defined by two quantities: digits and radices. `t` is a mixed-radix number that will contain terminal indices $\texttt{t}_\texttt{d}$ and corresponding radices $\texttt{t}_\texttt{r}$ equal to the number of terminals in each group. `pat` is a mixed-radix number that will contain terminal group indices $\texttt{pat}_\texttt{d}$, with $\texttt{pat}_\texttt{r}$ containing the corresponding radices. `i` is a cursor that points at our current position in `S`.)

P7. [Find matching terminal.] Find $\texttt{p}_\texttt{i}$ in the terminals of `N[i]`. If not found, go to step P4. Otherwise, update the mixed-radix numbers as follows. Let `j` be the index of $\texttt{p}_\texttt{i}$'s terminal group, and let `N[i][j]` represent that group. Perform the following assignments:

Set $\texttt{t}_\texttt{d}$`[i]` to the index of $\texttt{p}_\texttt{i}$ within `N[i][j]`.

Set $\texttt{t}_\texttt{r}$`[i]` $\leftarrow$ `|N[i][j]|`, which is the number of terminals in `N[i][j]`.

Set $\texttt{pat}_\texttt{d}$`[i]` $\leftarrow$ `j`.

Set $\texttt{pat}_\texttt{r}$ $\leftarrow$ `|N[i]|`, which is the number of terminal groups in `N[i]`.

Set `ids` to the union of itself with the source identifiers attached to $\texttt{p}_\texttt{i}$.

(For example, in Figure 4.10 on page 71, `superman` is produced by the L_8 nonterminal. It is the first terminal of the third terminal group, so its index is 0 within its terminal group, which has index 2. Therefore, if $\texttt{p}_\texttt{i} = \texttt{superman}$ and `N[i]` $= L_8$, we would set: $\texttt{t}_\texttt{d}$`[i]` $\leftarrow 0$; $\texttt{t}_\texttt{r}$`[i]` $\leftarrow 2$, since there are two terminals in its terminal group; $\texttt{pat}_\texttt{d}$`[i]` $\leftarrow 2$, for the third terminal group; and $\texttt{pat}_\texttt{r} \leftarrow 3$, since there are three terminal groups in L_8.)

P8. [Increment cursor.] Set `i` $\leftarrow$ `i` $+ 1$. If `i` $\geq$ `n`, go to step P9. Otherwise, go to step P7.

P9. [Compute probability.] Set:

$$\texttt{pr} \leftarrow \texttt{Pr[S]} \cdot \prod_{i=0}^{\texttt{n}-1} \texttt{Pr}\left[\texttt{N[i]}\left[\texttt{pat}_\texttt{d}\texttt{[i]}\right]\right]$$

where `N[i][j]` is the $\texttt{j}^{th}$ terminal group of the nonterminal `N[i]`.

(If we are here, the password has been parsed. Compute its probability based on the current structure and the terminal groups recorded in step P7. This is the same probability formula used to generate patterns in step M2 of Algorithm 4.4M.)

P10. Return a tuple containing: `pr`; a pattern identifier composed of `S` and `pat`; `t`; and source identifiers `ids`. (If multiple parses are possible, go to step P4 and return more tuples if found.) ▌

4.7.2 Accounting for pattern compaction

In this section, we describe the `PATTERN-OFFSET` algorithm. This algorithm was called in step A2 of Algorithm 4.7A and is required to compensate for pattern compaction.

The lookup table tells us how many guesses are needed to get to the start of a password's pattern. The quantity $\texttt{t}_{\texttt{GN}}$, computed in step A5 of Algorithm 4.7A, told us how many guesses are produced from the start of a password's pattern to a given

password. It is a simple matter to add these two numbers, and this would give us the password's guess number if we were not using pattern compaction. However, pattern compaction means the lookup table points at the first permutation of a password's pattern, and `PARSE` might return results for a different permutation. This means we need to compute an additional *compaction offset* that tells us how many guesses are between the two permutations. We also find the lexicographically first permutation, which is used for scanning the lookup table by step A3 of Algorithm 4.7A. Both of these values are computed by Algorithm C, given later, but first we describe the problem in more detail.

Multiple permutations

We make use of several functions in the computation of `PATTERN-OFFSET`. Since we did not develop these functions, we devote minimal space to describing them, and do so in Appendix A. In this section, we provide some context for why Algorithm C is so complex. We examine a small part of the algorithm—computing the rank of a pattern identifier.

In order to compute the compaction offset, we need to be able to rank a pattern identifier in some consistent order. Recall that a pattern identifier is composed of a structure and a mixed-radix number, such as:

$$D_3L_8D_3\langle 0,0,1\rangle$$

The numbers in the angle brackets are the digits of the mixed-radix number. Its radices are not shown explicitly but can be derived based on the PCFG and given structure.

Recall from Section 4.5 that pattern identifiers can be very complex, for example:

$$L_1D_1L_1D_1L_1D_1L_1D_1\langle 0,0,1,0,2,3,3,2\rangle \tag{4.7.1}$$

The latter identifier actually contains two permutations:

$$L_1D_1L_1D_1L_1D_1L_1D_1\langle 0,-,1,-,2,-,3,-\rangle \tag{4.7.2}$$
$$L_1D_1L_1D_1L_1D_1L_1D_1\langle -,0,-,0,-,3,-,2\rangle \tag{4.7.3}$$

We can provide either (4.7.2) or (4.7.3) to the function `PERM-RANK-M`, described in Appendix A.2, to receive a ranking of the given permutation within its space of possible permutations. Calling `PERM-RANK-M` on (4.7.2) would yield 0, since it is the first permutation when permutations are ordered lexicographically. It is also the only permutation whose digits are in nondecreasing order, so it is the

permutation that was recorded when generating the lookup table in step M6 of Algorithm 4.5M. It is always the case that we record the permutation with rank 0 in the lookup table. For (4.7.3), calling `PERM-RANK-M` would yield 1, since it is the second permutation in lexicographic order.[27]

This gives us a rank for each permutation, but we want a rank for the entire pattern identifier. We do this by assigning significance to the permutations from left to right. The leftmost permutation, (4.7.2) in our example, is the most significant. We create a mixed-radix number out of these permutation ranks: the ranks are the digits of the number, and the radices are the number of possible permutations for a given permutation. The radices are computed using the `PERMS-TOTAL` function described in Appendix A.1. To get the final rank of a pattern identifier, we convert this mixed-radix number to a single radix using the `SINGLE-RADIX` function given in Appendix A.3.

We now present Algorithm C which implements the `PATTERN-OFFSET` function.

Algorithm C (*Compute compaction offset and first permutation*). Given a Weir PCFG `G` and a pattern identifier `pat`, compute the compaction offset `c` and `pat`'s first permutation `fpat`.

Let $\texttt{pat}_\texttt{d}$ and $\texttt{pat}_\texttt{r}$ be the digits and radices of the mixed-radix number in `pat`, and let `S` be its structure. Let `n` be the number of nonterminals in `S` and let `N[0]`, ..., `N[n-1]` index those nonterminals from left to right.

C1. [Collect repeated nonterminals.] Scan `S` for repeated nonterminals and create an array `R` such that `R[0]` $=$ $\texttt{RN}_0$, `R[1]` $=$ $\texttt{RN}_1$, ..., where $\texttt{RN}_\texttt{i}$ are different repeated nonterminals in `S`. For each repeated nonterminal, we find the minimum index with which it appears in `S`, and ensure that `R` is sorted in ascending order by each repeated nonterminal's minimum index.
(We store the repeated nonterminals in `R` so we can refer to them later. We also store their order of appearance in `S` to provide a consistent ordering of permutations.)

C2. [Collect permutations.] Create an empty hash table called `PERMS`. For each repeated nonterminal in `R`, create an empty list `p`, then:
Scan `S` in order from `N[0]` to `N[n-1]`. Where `N[j] = R[i]`, append `pat[j]` to `p`. After `S` has been scanned, set `PERMS[R[i]]` $\leftarrow$ `p`.
(`PERMS` is populated with each repeated nonterminal in `S` and their associated permutations. For example, if `pat` were the mixed-radix number from (4.7.1), we would store `PERMS[`L_1`]` $\leftarrow$ `[0, 1, 2, 3]` and `PERMS[`D_1`]` $\leftarrow$ `[0, 0, 3, 2]`.)

[27] The first is $L_1D_1L_1D_1L_1D_1L_1D_1\langle -,0,-,0,-,2,-,3\rangle$.

C3. [Compute ranks and total number of permutations.] Let `k` be the length of `R`. Construct a mixed-radix number `ranks` of length `k`. Let $\texttt{ranks}_\texttt{d}$ and $\texttt{ranks}_\texttt{r}$ stand for the digits and radices of `ranks`, respectively, and let $\texttt{ranks}_\texttt{d}\texttt{[0]}$ and $\texttt{ranks}_\texttt{r}\texttt{[0]}$ point at the most significant digit and radix in `ranks`, respectively. For each position `i` in `0, ..., k` $-$ 1:

Set $\texttt{ranks}_\texttt{d}\texttt{[i]} \leftarrow$ `PERM-RANK-M(PERMS[R[i]])`.

Set $\texttt{ranks}_\texttt{r}\texttt{[i]} \leftarrow$ `PERMS-TOTAL(PERMS[R[i]])`.

(If there were only one repeated nonterminal `RN` in `S`, we could simply return `PERM-RANK-M(PERMS[RN])` and return. Since there can be many repeated nonterminals, however, and each one might have a different set of possible permutations, we use a mixed-radix number to represent them. We use their order of appearance in `S`, as previously collected in `R`, to determine the ordering of permutations from most to least significant.)

C4. [Compute rank across permutations.] Set `rank` $\leftarrow$ `SINGLE-RADIX(ranks)`. (This computes a rank for our pattern-identifier, across any number of permutations that it might contain.)

C5. [Compute guesses per permutation.] Let `N[i][j]` represent the $\texttt{j}^{th}$ terminal group of the nonterminal `N[i]` in `G`, and let `|N[i][j]|` represent the number of terminals in that group. We can compute the number of guesses covered by a single permutation of a pattern with:

$$\texttt{size} = \prod_{\texttt{i}=0}^{\texttt{n}-1} \texttt{|N[i][pat}_\texttt{d}\texttt{[i]]|}$$

C6. [Compute compaction offset.] Set `c` $\leftarrow$ `rank` $\cdot$ `size`.

C7. [Construct first permutation.] Set `fpat` $\leftarrow$ `pat`. For each nonterminal $\texttt{RN}_\texttt{i}$ in `PERMS`, sort its array values in ascending order, then replace the corresponding values in `fpat` with the values in $\texttt{PERMS[RN}_\texttt{i}\texttt{]}$.
(Recall that for the mixed-radix number $L_1D_1L_1D_1L_1D_1L_1D_1\langle 0,0,1,0,2,3,3,2\rangle$ (4.7.1) we would have stored `PERMS[`L_1`]` $\leftarrow$ `[0, 1, 2, 3]` and `PERMS[`D_1`]` $\leftarrow$ `[0, 0, 3, 2]`. We now sort these arrays, and rewrite `fpat`. So we have: `PERMS[`L_1`]` $\leftarrow$ `[0, 1, 2, 3]` and `PERMS[`D_1`]` $\leftarrow$ `[0, 0, 2, 3]`, which when written back to `fpat` yields `fpat` $= L_1D_1L_1D_1L_1D_1L_1D_1\langle 0,0,1,0,2,2,3,3\rangle$. Each of the permutations in `fpat` are now in nondecreasing order, which means they have rank 0. `fpat` is the pattern identifier recorded in the lookup table.)

C8. [Return.] Return `c` and `fpat`. ∎

As you can see, computing the compaction offset is complex and requires coordinating between multiple data structures: the string of nonterminals that

comprises the structure of the pattern, a mapping of nonterminals to their corresponding values in a pattern identifier, and several mixed-radix numbers. As we explained in Section 4.3.1, the advantage of pattern compaction increases as passwords with more complex structures are modeled.

Though this complexity pays off in reduced size of the lookup table, it introduces numerous opportunities for failure. In retrospect, another approach to reducing the size of the table, such as a compression scheme, might have been a better choice over pattern compaction in terms of time spent versus space saved.

Chapter 5

Improvements to the Guessing Model

A *guessing model* assigns probabilities to passwords. As such, it describes a particular distribution of passwords. The previous chapter introduced a number of algorithms used to assign guess numbers efficiently, but the underlying guessing model is the same model presented in Section 3.4 and introduced by Weir in 2009 [146]. In this chapter, we make a number of improvements to the guessing model to improve its guessing efficiency. Making these improvements is intended to give us a more accurate metric of password strength, given our threat model from Section 1.1.

Many of the improvements are inspired by techniques used in popular cracking tools, such as mask attacks and brute-forcing when dictionary entries are exhausted [54]. We make four major improvements: 1) learning string frequencies, 2) generating unseen terminals, 3) hybrid structures, and 4) linguistic tokenization. We perform experiments and use guessing curves throughout the chapter to evaluate these improvements.

5.1 Experiments

The improvements described in this chapter are evaluated in various experiments. Reproducing any of these experiments requires three things: the framework, parameters, and data. You can download the framework from `https://github.com/cupslab/guess-calculator-framework/releases/tag/v1.0.0` and we provide a list of requirements for running it in `https://github.com/cupslab/guess-calculator-framework/blob/v1.0.0/INSTALL.md`.

Each experiment has its own set of parameters and is identified in this thesis by a name, e.g., *Experiment 1*. Appendix B contains tables with these parameters,

organized by name. Each graph built from experimental results will contain the experiment names in its caption.

These parameters are encoded in configuration files that you can download from `https://cups.cs.cmu.edu/chili/projects/passwords-gcf-thesis-configs/wiki/Wiki`. We do not provide a link to the leaked password sets used in our experiments, or our datasets collected from Mechanical Turk, but instructions for preparing these datasets for use in experiments can be found in Appendix B.4. Datasets collected from Mechanical Turk are available for research purposes by contacting the author.

5.2 Learning string frequencies and quantization

We begin by examining the first improvement we made to the Weir 2009 guessing model: learning string frequencies. Weir's original system did not learn alphabetic string probabilities from training data. Instead, it assigned all alphabetic terminals a uniform probability. This is described by Weir as an "implementation convenience" [144]. It greatly decreases the number of terminal groups,[1] and this greatly reduces the memory required to store groups when strings are produced in probability order. This memory limitation is not a concern with the guess-calculator framework because we use intelligent skipping, as described in Section 4.4, to produce patterns above a particular cutoff out of order and sort them later.

Unfortunately, learning string frequencies still poses a significant challenge to the framework because we are still concerned with limiting the number of terminal groups. Using precise string frequencies reduces the advantage of storing patterns over explicit guesses, since terminal groups become much smaller. While this can increase guessing efficiency, it makes it infeasible for us to produce the hundreds of trillions of guesses that real adversaries are believed to generate [54].

To counteract this, we add a quantizer to the framework that can be used to reduce the number of unique terminal probabilities.[2] Figure 5.1 shows how quantization can limit the number of terminal groups. Quantization reduces accuracy. It changes empirically observed probabilities to different values, to increase the numbers of terminals that share probabilities. As we described in Section 4.3.1, decreasing the number of terminal groups increases the speed with which we can search the space of strings produced by the PCFG.

To mitigate the negative impact of quantization, we use a two-step procedure.

[1] Described in Section 4.3.1.

[2] The quantization module is shown in the context of the rest of the framework in Figure 4.2 on page 50.

Θ: Structures

$\theta_{\mathcal{S}\to L_8D_3} = 1.0$

Before quantization		After quantization	
Alphabetic terminals	Digit terminals	Alphabetic terminals	Digit terminals
$\theta_{L_8\to\text{password}} = 0.21$	$\theta_{D_3\to 123} = 0.520$	$\theta_{L_8\to\text{password}} = 0.20$	$\theta_{D_3\to 123} = 0.520$
$\theta_{L_8\to\text{baseball}} = 0.20$	$\theta_{D_3\to 000} = 0.280$	$\theta_{L_8\to\text{baseball}} = 0.20$	$\theta_{D_3\to 000} = 0.048$
$\theta_{L_8\to\text{iloveyou}} = 0.19$	$\theta_{D_3\to 111} = 0.060$	$\theta_{L_8\to\text{iloveyou}} = 0.20$	$\theta_{D_3\to 111} = 0.048$
$\theta_{L_8\to\text{princess}} = 0.16$	$\theta_{D_3\to 222} = 0.058$	$\theta_{L_8\to\text{princess}} = 0.15$	$\theta_{D_3\to 222} = 0.048$
$\theta_{L_8\to\text{sunshine}} = 0.14$	$\theta_{D_3\to 333} = 0.028$	$\theta_{L_8\to\text{sunshine}} = 0.15$	$\theta_{D_3\to 333} = 0.048$
$\theta_{L_8\to\text{superman}} = 0.10$	$\theta_{D_3\to 444} = 0.028$	$\theta_{L_8\to\text{superman}} = 0.10$	$\theta_{D_3\to 444} = 0.048$
$\theta_{L_8\to\text{whatever}} = 0.10$	$\theta_{D_3\to 555} = 0.013$	$\theta_{L_8\to\text{whatever}} = 0.10$	$\theta_{D_3\to 555} = 0.048$
	$\theta_{D_3\to 666} = 0.007$		$\theta_{D_3\to 666} = 0.048$
	$\theta_{D_3\to 777} = 0.004$		$\theta_{D_3\to 777} = 0.048$
	$\theta_{D_3\to 888} = 0.001$		$\theta_{D_3\to 888} = 0.048$
	$\theta_{D_3\to 999} = 0.001$		$\theta_{D_3\to 999} = 0.048$

Figure 5.1: Weir PCFG example with quantization. The square brackets to the left and right of the terminals columns denote *terminal groups*. A terminal group is a set of terminals that have the same probability and are produced by the same nonterminal (L_8 or D_3 in this case). Notice that the number of terminal groups after quantization, on the right, is greatly decreased compared to the naturally occurring terminal groups on the left. This allows patterns to represent a larger set of guesses. Alphabetic terminals were quantized using the Lloyd-Max algorithm with 3 levels requested. Digit terminals were quantized with two levels requested.

Nonterminals

At the nonterminal level, we use the Lloyd-Max quantization algorithm [87, 94]. Given a set of values to quantize and a given number of quantization levels, i.e., a desired number of terminal groups, the Lloyd-Max algorithm is a standard way to minimize the amount of error introduced by the quantization. Error is measured by the Mean-Squared Error (MSE) between the quantized and original values. Allocating more levels can decrease quantization error, but also decreases the efficiency of our framework.

A natural consequence of attempting to minimize MSE is that large probability values are not quantized. In other words, they are not altered by the algorithm. Changes to large probability values affect MSE more than changes to small ones, so

the algorithm tries to avoid doing this. Consider the digit terminals in Figure 5.1. While the intuitive choice might be to cluster the two large values, 0.520 and 0.280, into one cluster, the Lloyd-Max algorithm does not do this. Instead, it leaves the 0.520 value alone and clusters the remaining values. This is the optimal solution for minimizing quantization error, since disturbances to large probability values have more impact than disturbances to small ones.

The Lloyd-Max algorithm is not guaranteed to find an optimum, so we run it 1,000 times with random initialization values. After all iterations are complete, we take the quantization with the lowest MSE. As explained in Section 4.6, we run these iterations in parallel across multiple cores in the host machine.

Structures

The structure level is where we decide how many quantization levels to allocate to a given nonterminal. Assume we have already chosen some overall number of levels to allocate across the entire grammar. We assign levels from this global pool to each nonterminal based on their probability of appearance across structures. This is similar in spirit to how the Lloyd-Max algorithm minimizes the error on high-probability items; by allocating more levels to high-probability nonterminals, we hope to reduce error. In testing, we found that this worked far better than a naïve assignment of a fixed number of levels to each nonterminal. The Lloyd-Max algorithm is run for each nonterminal, using its assigned number of levels.

Next, we discuss how we choose the overall number of levels. Figure 5.2 shows the total distortion over a range of levels using this method with a `basic8` policy.[3] Based on these results, we use a minimum of 500 quantization levels for all experiments since this seems to provide a good trade-off between speed (fewer levels) and accuracy (more levels).

Our allocation strategy works in practice but has a number of shortcomings. For example, a high probability nonterminal might have just a single production rule, so assigning more than one level to this nonterminal is wasteful, when these levels could be allocated to other nonterminals. Our algorithm does not take this into account. We also have not tested different strategies for allocation of levels, beyond a fixed number of levels per nonterminal and simple, probability-weighted allocation. We do not address these shortcomings in this thesis.

[3] Our convention for naming policies is described in Section 2.4.4.

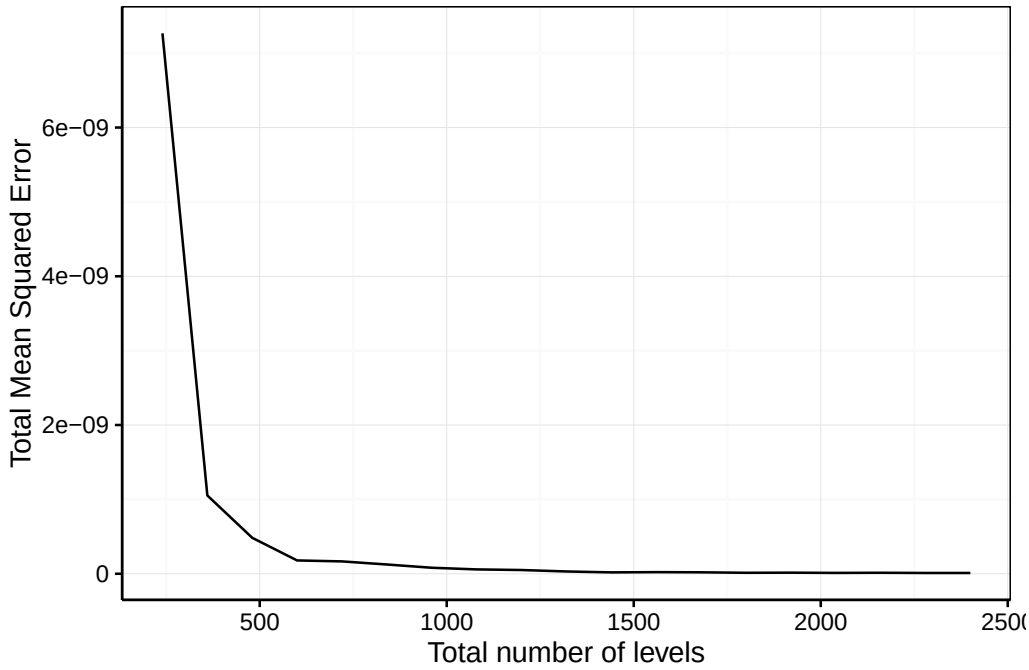


Figure 5.2: Total MSE (mean squared error) for a range of global levels from 120 to 2,400. More quantization levels decrease the efficiency of the guess-calculator framework. Notice the elbow in the graph which indicates a good trade-off between levels and accuracy.

5.3 Producing unseen strings

The next improvement to the guessing model is the ability to guess terminals that were not seen in the training data. We have found that cracking tools like John the Ripper and Hashcat are better than the guess-calculator framework at guessing passwords that are all digits, because they can brute force all-digit strings. In contrast, our PCFG approach is constrained to those terminals in the grammar, i.e., those terminals in the training data.

To address this issue, we added the ability to produce *unseen* terminals to the guess-calculator framework. We do this by adding a special symbol to the grammar that can represent a group of unseen terminals. Figure 5.3 shows the result of using this improvement on a `basic8` policy.

When this improvement is enabled, the framework uses a "Good-Turing estimator" [52] during the learning phase to assign a probability to unseen terminals.[4] To provide an intuition into how this works, we present a simple example. Details about our implementation will be provided later.

[4]Good-Turing estimation is one of a general family of techniques known as *smoothing*, which model the probabilities of unseen events. For a survey of these techniques see [27].

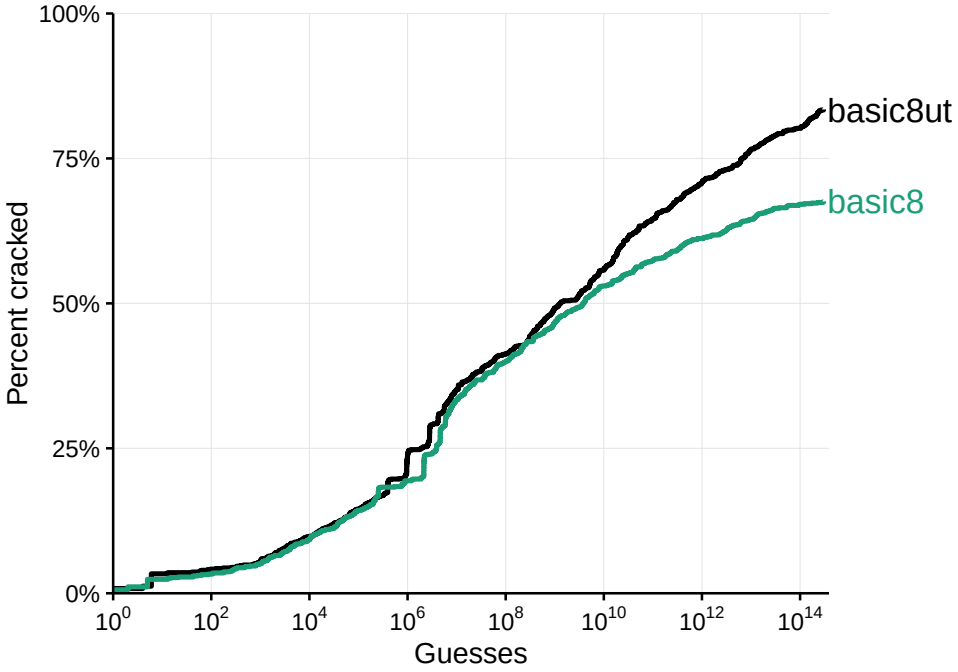


Figure 5.3: Experiment 2 — Evaluating unseen terminal generation for a basic 8-character policy. The `basic8ut` curve shows the performance of the guess-calculator framework with unseen terminal production enabled. The `basic8` curve starts to plateau after 10^{10} guesses as it runs out of strings in its training data and moves to less successful structures.

Say the training data contains 5,500 of the 10,000 possible four-digit numbers. This means that there are 4,500 four-digit numbers that will never be guessed by the grammar. These are *unseen terminals*, and we want to estimate their probability. A Good-Turing estimator does this by using the probability of *singletons* as a guide. Singletons are items that have only been seen once.

Imagine that we could collect one more sample of a four-digit number. Let us call this sample `s`. There is some probability `X` that `s` is a number we have not seen before. If that happens, we now have a new number `s` that we previously had not seen before, but have now seen once. Therefore, `s` is now a singleton! To estimate the probability of seeing a new singleton, we use the total probability mass of existing singletons: if 10% of the samples we have seen turned out to be singletons (as opposed to being seen more often), we estimate the probability `X` of any new singleton as 10%. Since any new singleton is a previously unseen terminal, this means that we estimate the total probability of unseen terminals as 10%.

Of course, the observed probabilities for all seen terminals already adds up to 100%. To make space for the unseen terminal group, we multiply the existing probabilities by 90% and allocate the remaining 10% for unseen terminals. We add the unseen terminals to the grammar as a new terminal group that has a

probability of $10\%/4500 = 0.00222\%$ per terminal. For simplicity, we assume that all unseen terminals have the same probability, so they can go into a single terminal group.[5]

5.3.1 Good-Turing estimation within the guess-calculator framework

Unfortunately, our implementation of Good-Turing estimation is not as straightforward as the previous example. In this section, we describe the issues that we encountered and how we chose to deal with them. The rules that follow are conservative and intended to ensure that the framework is robust in the presence of difficult inputs, but they might result in unseen-terminal probabilities that are much lower than they should be. These decisions were made out of expedience and are certainly not optimal. The ability to guess unseen terminals at all constitutes a huge improvement to the framework, as already shown in Figure 5.3, and just getting it to work was our main focus. Changes to the following rules, or our unseen-terminal estimation process in general, might yield better guessing models.

It is also worth noting that unseen-terminal estimation happens before the quantization step from Section 5.2. The unseen terminal group is marked and excluded from quantization.

No singletons

In Section 4.2, we described how a user can specify many different data sources and weightings and the guess-calculator framework will combine them into a single corpus. Unfortunately, this makes it unclear what should be considered a singleton. If a singleton is defined as an item with frequency 1, then it is entirely possible for the corpus to have no singletons if every source was assigned a fractional weight, or a weight larger than 1. Similarly, singletons might appear in the corpus, but come from small, minor datasets with weight 1, while the datasets we care about, which are assigned larger weights, do not have their singletons counted.

It is also possible for a dataset not to have singletons, yet have unseen items. This can happen by chance, or when items below a particular frequency are excluded from the dataset.[6] We believe it is incorrect not to assign unseen terminals in this case.

[5] This does not hold across nonterminals. The unseen terminals for three and five-digit strings can have different probabilities than unseen four-digit strings. Each terminal list is handled separately.

[6] This is the case with the Google Web Corpus [15].

```
‘ ~ ! @ # $ % ^ & * ( ) - _ = + [ ] { } | ; :  ’  " , .  < > / \ ?
```

Figure 5.4: Symbol set for unseen terminal generation

To account for these situations, we take as singletons the items with the most common frequency. This ensures that some unseen-terminal probability is assigned. In our experience, this tends to be a good proxy for the items that we care about.

Fixed domain

In our application, we know the total number of possible terminals. For example, if we are dealing with four-digit strings, we know that there are 10,000 of them. This leads to some unexpected situations. For example, if we have seen 9,999 four-digit strings, and 1,000 of them are singletons, the Good-Turing estimator for unseen strings is ≈10%. This would give the remaining unseen string a probability of 10%, since there is only one unseen string remaining! To account for this, we cap the probability of an unseen terminal to 80% of the probability of a seen terminal. If the Good-Turing estimator comes out higher than this, we cap it to the lower amount and adjust the total unseen-probability mass (and the amount that we adjust the existing probabilities) accordingly. The 80% value was chosen arbitrarily.

Symbol generation

Though it is trivial to brute-force alphabetic and digit strings, symbol strings are more difficult because there are a large space of potential symbols to try. We chose to brute-force symbol strings using only a subset of the possible symbols, taken from the set in Figure 5.4. These are the printable symbols on a US keyboard. This avoids wasting guesses on symbols that are rarely encountered, but it means that we cannot brute-force many symbol strings, such as unseen UTF-8 sequences.

5.3.2 Assigning guess numbers to unseen strings

From the perspective of the high-level modules of the guess-calculator framework shown in Figure 4.2 on page 50, terminal groups containing unseen terminals are treated no differently from any other terminal groups. The number of strings they contain can be determined based on the size of the candidate terminal space and the number of seen terminals. For example, if the unseen terminal is a 6-character alphabetic string, and we have seen 1,500 strings of this type, then the size of the unseen terminal group is $26^6 - 1{,}500 = 308{,}914{,}276$. This makes it easy to determine the total number of guesses represented by a pattern that includes

unseen terminals, so our intelligent skipping and pattern compaction algorithms require no changes.

When we look up a password containing unseen terminals, however, we need to determine its rank within its pattern as part of determining its guess number. This is required by the PARSE function implemented by Algorithm 4.7.1P. We make a small modification to step P7 of Algorithm 4.7.1P when unseen terminals are involved. Instead of failing when a terminal is not found, we invoke Algorithm U below to determine an unseen-terminal rank, if possible.

Algorithm U computes the rank of an unseen terminal by first computing the terminal's lexicographic rank in the space of terminals with the same composition, and then subtracting the number of seen terminals that come before it. It uses the SINGLE-RADIX function presented in Appendix A.3 to compute the rank.

Algorithm U (*Rank unseen terminal*). Given a terminal $\mathtt{p_i}$ that could be produced by a nonterminal N[i] in a Weir PCFG G. Return failure or the index of $\mathtt{p_i}$ within N[i][j], where j is the unseen terminal group produced by N[i].[7]

Let n be the length of $\mathtt{p_i}$.

U1. [Check for symbols outside our chosen set.] Scan $\mathtt{p_i}$ for symbols (characters that are neither alphabetic nor digits) outside of the set shown in Figure 5.4. If any are found, return failure. (We can only brute-force terminals with alphabetic characters, digits, or symbols from that set.)

U2. [Initialize mixed-radix number.] Let r be a mixed-radix number of length n with digits $\mathtt{r_d}$ and radices $\mathtt{r_r}$. For each position k in r, set r to the following values:

If $\mathtt{p_i}$[k] is alphabetic:
  Set $\mathtt{r_r}$[k] $\leftarrow$ 26.
  Set $\mathtt{r_d}$[k] $\leftarrow$ $\mathtt{p_i}$[k] $-$ '*a*'.
If $\mathtt{p_i}$[k] is a digit:
  Set $\mathtt{r_r}$[k] $\leftarrow$ 10.
  Set $\mathtt{r_d}$[k] $\leftarrow$ $\mathtt{p_i}$[k] $-$ 0.
If $\mathtt{p_i}$[k] is a symbol:
  Set $\mathtt{r_r}$[k] $\leftarrow$ |SYM|.
  Set $\mathtt{r_d}$[k] $\leftarrow$ SYM.indexof($\mathtt{p_i}$[k]).

(This step sets the digits and radices of the mixed-radix number to the appropriate values based on the character class and value of $\mathtt{p_i}$[k]. Note that $\mathtt{p_i}$ is assumed to be lowercase. This will be discussed in Section 5.4.1. SYM

[7] Since the unseen terminals are constrained to have lower probability than any seen terminals, they always comprise the last terminal group produced by a nonterminal.

is the array of characters from Figure 5.4, and the `indexof()` method returns the position of the given character in the array, where 0 is the first position.)

U3. [Compute lexicographic rank.] Set $\texttt{rankp}_\texttt{i} \leftarrow \texttt{SINGLE-RADIX(r)}$.
(You can think of the previous step as bookkeeping. We categorized each character in $\texttt{p}_\texttt{i}$ and set `r` appropriately. This step distills that process into a single rank.)

U4. [Count seen terminals.] For each seen terminal `t` produced by `N[i]`, check step U1. If step U1 fails, ignore that terminal and move to the next. Set `count` to the number of seen terminals that come before $\texttt{p}_\texttt{i}$ in lexicographic order.

U5. [Compute corrected rank.] Return $(\texttt{rankp}_\texttt{i} - \texttt{count})$. ∎

As a simple example of Algorithm U, consider the following. The terminal `abcd` has lexicographic rank 731 (`aaaa` is rank 0.) We iterate over all seen terminals, counting up all terminals with lower rank, and subtract this number from the lexicographic rank. We can recognize these terminals because they come before `abcd` in lexicographic order. For example, `aaaz`, `abau`, `abcc`, etc. If 100 four-letter strings were seen that come before `abcd`, the rank of `abcd` returned by Algorithm U would be $731 - 100 = 631$. This is because once the guessing model reaches the unseen-terminal group of `N[i]`, the 631^{st} guess would be `abcd`. All of the seen terminals will have been part of previous guesses so we do not want to reproduce them again.

5.4 Using a more complex grammar

The original grammar of Weir et al. used a restricted set of nonterminals which only produced terminals of a single character class. For instance, the nonterminal D_4 produces only four-digit strings, nonterminal L_4 produces strings of 4 letters, etc.

We first describe our modifications to the grammar and then discuss general issues with making models more complex.

5.4.1 Uppercase nonterminal

We first modified the grammar to account for uppercase characters in the set of nonterminals. This changes the set of nonterminals Q described in Section 2.3.1 into $Q = \{\{U, L\}^i, D_i, S_i\}$. For example, where the terminal `Password` would correspond to the nonterminal L_8 in the original grammar, it corresponds to the nonterminal $(U_1 L_7)$ in our framework. This is a single symbol that represents

terminals where the first letter is uppercase and the next seven letters are lowercase, e.g., `Password`.

To implement this modification, we store terminals in lowercase but keep track of their case in nonterminals. We store a single list of L_8 nonterminals that is updated regardless of case. So if `password`, `Password`, and `PaSsWoRd` are observed, we count it as seeing the terminal `password` three times, and nonterminals L_8, (U_1L_7), and $(U_1L_1U_1L_1U_1L_1U_1L_1)$ each once. This allows us to compose strings like `BaSeBaLl` even if we have seen them in a different case and attach probabilities to different casing patterns.

We made this modification based on previous observations. Through surveys and examination of user-created passwords under complex policies, we found that case in passwords is usually used to comply with requirements rather than to convey meaning [76, 127]. In other words, the case of a word is usually not tied to the specific word being used. Therefore, we felt that encoding this information in the structure made more sense, because it allows any terminal to be uppercased. Anecdotally, we have found that this modification is a positive improvement. However, we have never formally evaluated it, so it is possible that it is less efficient than the original model.

5.4.2 Mixed-class nonterminals

The next modification to the grammar involves allowing nonterminals to represent a mix of character classes. Recall our restricted set of nonterminals $Q = \{\{U, L\}^i, D_i, S_i\}$. This restriction causes nonterminals to map only to terminals in which all characters have the same character class: alphabetic strings, digits, or symbols.

Since we are using a probabilistic context-free grammar, any relationship between terminals produced by different nonterminals is lost. This means that we cannot learn, or reproduce, any relationship between parts of a password that change character class. Under the original Weir grammar, when `password123!` is encountered in the learning phase, it must be learned as three separate terminals: `password`, `123`, and `!`. The fact that these strings might appear together is lost. If `password123!` is the most probable password in our training data, this is a serious failure of the model. This is discussed in more detail in Section 5.5.

To address this issue, we would like a PCFG that can produce strings like `password123!` with high probability. The simplest way to do this is to allow

`password123!` to be a single terminal.[8] To fit such a terminal into the Weir model, we greatly expand the set of possible nonterminals to $Q = \{\{U, L, D, S\}^i\}$, where i is still the string length of replacement terminals. For example, the single nonterminal corresponding to `password123!` can be represented by $(L_8 D_3 S_1)$. This nonterminal is replaced by all strings of eight letters, three digits, and one symbol found in the training data, and optionally unseen terminals of this form as well. This is used in the *hybrid structures* and *unsupervised tokenization* improvements presented in Section 5.5, but could potentially be used with more sophisticated learning algorithms.

5.4.3 Increasing model complexity

The original Weir grammar defined a relatively small set of possible nonterminals $Q = \{L_i, D_i, S_i\}$, where L represents alphabetic strings, D digits, S symbols, and i the string length of replacement terminals. Note that, for practical reasons, there is a cap on the maximum string length. In our implementation, we set the maximum string length, `M`, to 40, so the size of the set of possible nonterminals is $|Q| = 3 * T_{40} = 2460$.[9] This quantity is $O(\mathtt{M}^2)$. When we added mixed-class nonterminals to the grammar, we greatly expanded the set of possible nonterminals to $|Q| = \sum_{i=1}^{\mathtt{M}} 4^i \approx 10^{24}$. This is $O(4^{\mathtt{M}})$, i.e., it is exponential in the size of strings.

Adding complexity to the grammar, as with any model, is a double-edged sword. We might improve our ability to capture fine-grained trends in the data, but this can result in a model that does not generalize as well as a simpler model. For a trivial example, consider the addition of uppercase to the set of nonterminals as described in the Section 5.4.1. Suppose that our input data consists of two passwords: `Password12` and `12passworD`. Because the new model captures the dependency between uppercase letters and password structure, the model cannot generate any other passwords besides `Password12` and `12passworD`. In the original Weir model, however, this input creates a model that can generate two additional passwords: `passworD12` and `12Password`. The probabilities of the original passwords have to be reduced to make room in the model for the additional guesses.

Is this better? It depends. If the true policy contains the additional passwords, then the Weir model better represents the passwords of the true policy. Given a larger and more representative sample, our model would learn these additional

[8] Another solution is to retain `password`, `123`, and `!` as separate terminals, but have them produced by dedicated nonterminals that appear together in higher-level production rules. Techniques such as this were discussed in Section 2.7. In this thesis, we try the simple approach first and leave more complex approaches to future work.

[9] $T_i = \frac{i(i+1)}{2}$ is the sum of numbers from 1 to i.

passwords and assign them appropriate probabilities, but the size of a sample cannot be guaranteed. Conversely, it might be the case that the additional passwords generated by the original Weir model do not correspond to high-probability passwords in the true policy, in which case we reduced probabilities unnecessarily to make room for them.

Typically, there is some amount of training data beyond which complex models perform better than simple models. In this work, we provide some evidence for or against the various improvements that we have made. A more thorough investigation of these improvements, given substantially more data or other datasets, remains a topic for future work.

5.5 Tokenization

Allowing nonterminals to represent an arbitrary mix of character classes makes the grammar more powerful, because it increases the grammar's ability to learn from the data. Recall our previous example where `password123!` is the most common password. Weir's original implementation learns this as `password`, `123`, and `!`. These three elements are called *tokens* and the splitting of the original string into parts is known as *tokenization*.[10]

Because nonterminals in Weir's grammar are specified based on a single character class, tokenization has to take place at character-class boundaries. Our improved PCFG is more flexible, so it can contain the nonterminal $(L_8D_3S_1)$, for which `password123!` is the highest-probability terminal. By encoding `password`, `123`, and `!` together within a single terminal, we retain the information that they occur together. This information is known as a *dependency*.

Learning dependencies is important, especially for high-probability passwords. Weir's implementation is unable to learn the fact that these particular tokens all occur together, so it is unable to put these tokens back together with high probability.[11] By learning this dependency, we have the potential to learn more efficient guessing models.

We can look back at the original Weir PCFG as an instance of our expanded grammar that uses a *character-class tokenizer*. We call such a tokenizer a *Weir tokenizer*, since Weir et al. [146] had the original insight to tokenize passwords in this way. To learn more complex grammars, we can insert other tokenizers into the guess-calculator framework.

[10] See `https://en.wikipedia.org/wiki/Tokenization_(lexical_analysis)` or `http://opennlp.apache.org/documentation/1.6.0/manual/opennlp.html#tools.tokenizer.introduction`.

[11] We discuss this issue further in Sections 5.5.1 and 7.2.1.

We will see that this is an open problem. Though we have expanded the grammar with mixed-case nonterminals (and one could imagine further expansions) learning a complex grammar is difficult. Intuitively, this makes sense. The Weir tokenizer can only parse a given password in a single way, but a mixed-class tokenizer has an exponential number of tokenizations to choose from—any partition of a string is a potential tokenization. Choosing the best tokenization from among these alternatives is hard. The unsupervised approach of Berg-Kirkpatrick et al. [3] provides a potential solution to this, but we were unable to apply to our domain successfully. This is described in Section 5.5.3.

5.5.1 Hybrid structures

An easy way to capture dependencies within passwords is to learn whole passwords as single tokens. For example, if we encounter `password123!` as a password, we add the production rules: $\mathcal{S} \rightarrow (L_8 D_3 S_1)$ and $(L_8 D_3 S_1) \rightarrow$ `password123!` to our grammar. We call this an *untokenized structure* and such structures are both trivial to learn and work well to reproduce passwords from the training data with accurate probabilities.

This solves the problem of reproducing training data which has high-probability mixed-class passwords. Unfortunately, if we only learn untokenized structures, we completely lose the ability to generate strings that were not seen in the training data. To address this, we adopt a hybrid approach: learning untokenized and tokenized structures simultaneously. Perhaps surprisingly, this automatically favors untokenized structures over tokenized ones. Strings produced by tokenized structures tend to have much lower probability than untokenized ones, because more tokens leads to more probabilities being multiplied. This provides the nice property that early guesses match the most probable passwords in our training set. If desired, we could weight tokenized structures even lower than untokenized ones, so that low-probability passwords from the training set are favored over high-probability tokenized passwords. For this thesis, we stick with equal weighting, but the results from Section 5.5.3.2 will suggest that an additional penalty on tokenized structures would improve performance.

To illustrate a PCFG that includes hybrid structures, we reuse the example from Figure 2.1 in which our training data consists solely of the four passwords: `password!`, `password!`, `baseball!`, and `baseball123!`. The resulting PCFG under a hybrid approach can be seen in Figure 5.5 and the resulting guesses in Table 5.1. As shown in Table 5.1, we can still produce the unseen guess `password123!`, but passwords from the training set are preferred. This is because they additionally derive from untokenized structures.

Σ: {password!, baseball!, baseball123!, password, baseball, 123, !} (terminals)

$\mathcal{N}$: $\{(L_8S_1), (L_8D_3S_1), L_8, D_3, S_1\}$ (nonterminals)

$\mathcal{R}$: (structures)
$\mathcal{S} \to (L_8S_1)$
$\mathcal{S} \to (L_8D_3S_1)$
$\mathcal{S} \to L_8S_1$
$\mathcal{S} \to L_8D_3S_1$
(terminal productions)
$(L_8S_1) \to$ password!
$(L_8S_1) \to$ baseball!
$(L_8D_3S_1) \to$ baseball123!
$L_8 \to$ password
$L_8 \to$ baseball
$D_3 \to 123$
$S_1 \to$!

Θ: (probabilities)
$\theta_{\mathcal{S} \to (L_8S_1)} = 0.375$
$\theta_{\mathcal{S} \to (L_8D_3S_1)} = 0.125$
$\theta_{\mathcal{S} \to L_8S_1} = 0.375$
$\theta_{\mathcal{S} \to L_8D_3S_1} = 0.125$
$\theta_{(L_8S_1) \to \text{password!}} = 0.66\overline{6}$
$\theta_{(L_8S_1) \to \text{baseball!}} = 0.33\overline{3}$
$\theta_{(L_8D_3S_1) \to \text{baseball123!}} = 1.0$
$\theta_{L_8 \to \text{password}} = 0.5$
$\theta_{L_8 \to \text{baseball}} = 0.5$
$\theta_{D_3 \to 123} = 1.0$
$\theta_{S_1 \to !} = 1.0$

Figure 5.5: PCFG under the hybrid approach—both tokenized and untokenized structures—produced from a corpus that consists solely of the passwords: `password!`, `password!`, `baseball!`, and `baseball123!`. Compare with Figure 2.1 on page 22.

Guess #	Guess	Probability	Probability derivation
1	password!	0.4375	$\theta_{\mathcal{S}\to(L_8S_1)} \cdot \theta_{(L_8S_1)\to\text{password!}}$ $\theta_{\mathcal{S}\to L_8S_1} \cdot \theta_{L_8\to\text{password}} \cdot \theta_{S_1\to!}$
2	baseball!	0.3125	$\theta_{\mathcal{S}\to(L_8S_1)} \cdot \theta_{(L_8S_1)\to\text{baseball!}}$ $\theta_{\mathcal{S}\to L_8S_1} \cdot \theta_{L_8\to\text{baseball}} \cdot \theta_{S_1\to!}$
3	baseball123!	0.1875	$\theta_{\mathcal{S}\to(L_8D_3S_1)} \cdot \theta_{(L_8D_3S_1)\to\text{baseball123!}}$ $\theta_{\mathcal{S}\to L_8D_3S_1} \cdot \theta_{L_8\to\text{baseball}} \cdot \theta_{D_3\to 123} \cdot \theta_{S_1\to!}$
4	password123!	0.0625	$\theta_{\mathcal{S}\to L_8D_3S_1} \cdot \theta_{L_8\to\text{password}} \cdot \theta_{D_3\to 123} \cdot \theta_{S_1\to!}$

Table 5.1: Guesses produced by the PCFG of Figure 5.5. Compare with Table 2.1 on page 23.

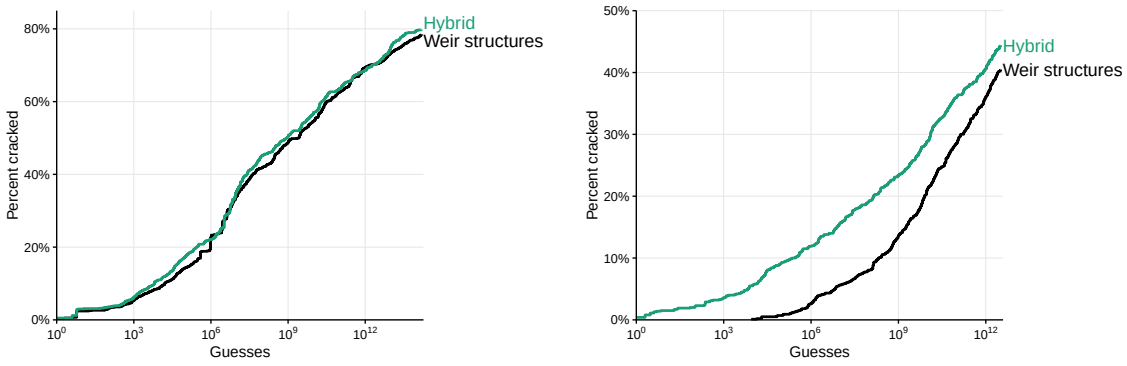
Guess	Empirical	Weir tokenized	Untokenized	Weir hybrid
password!	0.5	0.375	0.5	0.4375
baseball!	0.25	0.375	0.25	0.3125
baseball123!	0.25	0.125	0.25	0.1875
password123!	—	0.125	—	0.0625

Table 5.2: Comparison of probabilities for strings produced by PCFGs trained using various schemes. with a corpus that consists solely of the passwords: `password!`, `password!`, `baseball!`, and `baseball123!`. Empirical probabilities are the observed probabilities in the input corpus. Weir-tokenized probabilities are the result of character-class tokenization (values from Table 2.1). Untokenized probabilites are based on learning untokenized structures only. The Weir hybrid approach learns both untokenized and character-class-tokenized structures with equal probability (values from Table 5.1).

Probabilities for strings produced by PCFGs, trained using various schemes, are summarized in Table 5.2. The untokenized probabilities exactly match the empirical probabilities from the training data, but no other strings can be generated. In contrast, a Weir-tokenized PCFG can produce new guesses, but loses information about which whole strings were present in the training data. The hybrid approach averages the empirical and Weir-tokenized probabilities. Thus, it balances the need to reproduce high-probability strings from the training data with the need to generate new strings.

Evaluation

We illustrate the hybrid approach on two policies: `basic8` and `4class8`. The `basic8` policy requires an 8-character minimum length, but has no character-class requirements. The `4class8` policy also requires an 8-character minimum length, but also requires that the password contain uppercase, lowercase, digit, and symbol characters, at least one of each. Figure 5.6 shows how cracking performance using hybrid structures compares with Weir-tokenized structures on these policies. As



(a) `basic8` (b) `4class8`

Figure 5.6: Experiment 3A (`basic8`) & Experiment 3B (`4class8`) — Each graph shows guessing curves comparing the *hybrid structures* approach developed in this thesis and Weir-tokenized structures. Hybrid structures are significantly better at cracking mixed-class passwords, particularly in the early part of the curve crucial to online guessing, but do not help with cracking basic passwords. PCFGs were trained using public data and tested on a random holdout set of 1,000 RockYou passwords that conformed to each policy. Other improvements of the guess-calculator framework, such as string frequencies and unseen terminal generation, are enabled for all curves.

shown, hybrid structures have significantly better cracking performance than Weir-tokenized structures when cracking `4class8` passwords ($p < 0.001$ measured with G^1 test).[12] The difference between the models is particularly evident in the early part of the guessing curve.

While cracking `basic8` passwords however, performance is not significantly better ($p = 0.162$, G^1 test). This is likely because high-probability passwords are often composed of a single character class in `basic8`. The highest-probability passwords are shown in Table 5.3. This means they are not tokenized by a Weir-tokenizer, so their probabilities are not penalized by the resulting guessing model. This is in contrast to `4class8` passwords, which are split into a minimum of three chunks by a Weir-tokenizer (recall that case is ignored when tokenizing passwords and is retained only in structures.) The *tokenization penalty* is stronger on passwords with multiple character-class transitions, such as the highest probability `4class8` password: `P@ssw0rd`, which is split into five chunks.

5.5.2 Linguistic tokenization

Besides tokenizing passwords based on strings of the same character class, another way to tokenize passwords is to try to divide them into linguistic elements,

[12] This statistical test is explained in Section 6.1.4.

1	123456789	18	elizabeth	35	princess1
2	password	19	tinkerbell	36	carolina
3	iloveyou	20	samantha	37	alejandro
4	princess	21	danielle	38	brittany
5	12345678	22	jonathan	39	alejandra
6	babygirl	23	987654321	40	patricia
7	michelle	24	computer	41	tequiero
8	sunshine	25	whatever	42	stephanie
9	chocolate	26	spongebob	43	blink182
10	password1	27	softball	44	victoria
11	butterfly	28	princesa	45	fernando
12	liverpool	29	alexandra	46	cristina
13	football	30	estrella	47	babygurl
14	jennifer	31	beautiful	48	baseball
15	1234567890	32	poohbear	49	greenday
16	superman	33	alexander	50	november
17	basketball	34	christian		

Table 5.3: 50 most popular `basic8` passwords in the RockYou password set. Most passwords are composed of only one character class (all letters or all digits).

Password	Ranking	Linguistic Tokenization
iloveyou	5	i love you
lovely	15	lovely
iloveu	22	i love u
loveme	38	love me
loveyou	52	love you

Table 5.4: This table demonstrates how tokenization can find common semantic elements within passwords. The top five passwords containing the substring "love" from the RockYou dataset are shown along with their ranking in the overall dataset and the result of running these passwords through our linguistic tokenizer.

such as words. Passwords seem to be composed of linguistic elements (see Section 2.4.3), and tokenizing passwords based solely on character class does not reveal these elements. For example, Table 5.4 shows the five most frequent passwords containing "love" from the RockYou dataset. The string "love" is very common in passwords, but a Weir tokenizer would never learn this fact because it parses passwords based solely on character class. All of the passwords in Table 5.4 are treated as single tokens by simple parsers, because they have no way to break them down further.

We use the term *unbroken* for text that has had the spaces between words removed. We can *break* long strings of unbroken text using a linguistic model built on text with spaces. An *n-gram*, in this context, is a string of n broken words. When strings of text are found in passwords, we can break them by identifying unbroken n-grams in the text. For example, if we are aware of the n-gram "i love you," we can remove the spaces from it and identify the token `iloveyou` in passwords. If found, we can break it into the separate tokens `i`, `love`, and `you`.

We built a simple linguistic tokenizer to do this using the Google Web Corpus (GWC) as a dataset. The GWC was published in 2006 and contains "English word n-grams and their observed frequency counts" [15] based on a corpus of one trillion words from web pages. Only n-grams with frequencies above 40 were kept by the publishers and the maximum n-gram length is five—this means the GWC contains sequences of five words or less. Unfortunately, the GWC is only a good tokenization source for strings of letters and digits. Strings of symbols are tokenized so that each symbol is a separate token, and digits are tokenized separate from letters. Using this tokenizer, we can find common words in passwords and tokenize based on word boundaries.

Linguistic models The tokens that we discover using our linguistic tokenizer are still simple letter strings, e.g., "iloveyou" would get tokenized to an $L_1L_4L_3$ structure. Recently, other researchers have looked at more complex linguistic tokenization including part-of-speech tagging and name identification, which greatly increases the variety of nonterminals in the PCFG [140]. We do not evaluate these models in this work.

Figure 5.7 shows how the use of linguistic tokenization improves cracking for large guess numbers. Just as Weir tokenization allows us to guess passwords we have not seen before by combining terminals in new ways, linguistic tokenization allows us to combine words in new ways to produce new guesses. Conversely, breaking passwords up into many tokens imposes a tokenization penalty that decreases guessing performance for low guess numbers compared with Weir tokenization. This effect can be mitigated through the use of hybrid structures, and we explore this issue in Section 7.3.

Our approach

We first provide a high-level overview of our linguistic tokenizer and then provide more detail on the data structures that it uses. The algorithm is greedy and we do not claim that its behavior is optimal, but we find that it works well for policies with long passwords, as shown in Figure 5.7.

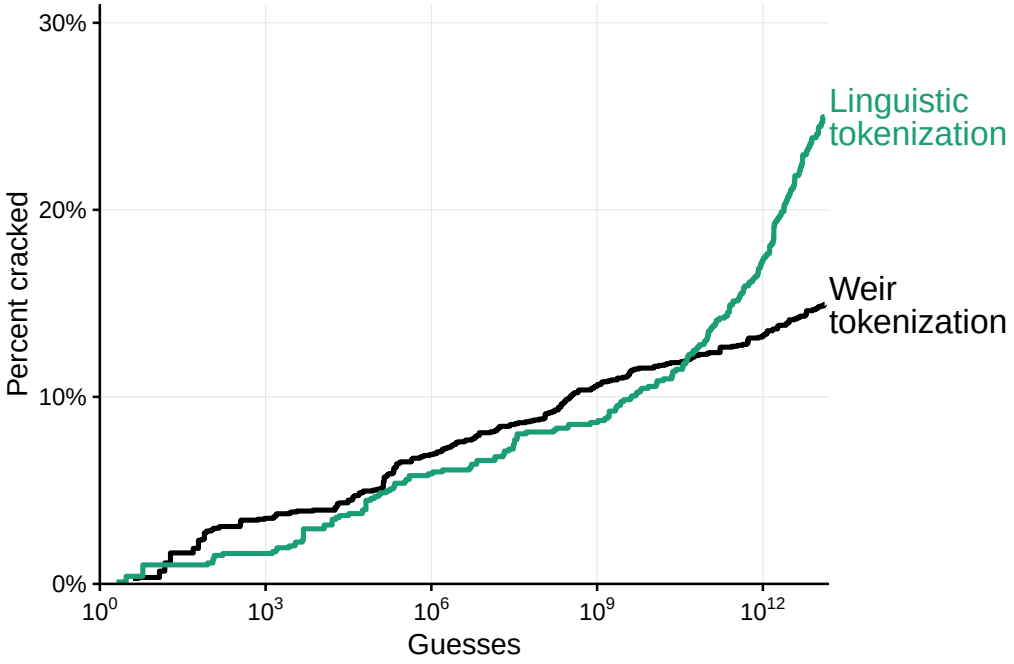


Figure 5.7: Experiment 4 — Guessing curves for a basic 16-character policy. Using linguistic tokenization cracks many more passwords as guess numbers grow large, but is slightly worse in early guessing.

Current string	Tokenized prefix	Tokenized string
correcthorsebatterystaple	**correct** horse	correct
horsebatterystaple	**horse** battery	correct horse
batterystaple	**battery**	correct horse battery
staple	**staple**	correct horse battery staple
thenoodles	**the** noodles	the
noodles	**noodles**	the noodles

Table 5.5: Example of our tokenizer operating on the string `correcthorsebatterystaple`, which does not appear in the Google Web Corpus, and `thenoodles`, which appears 77,281 times. Accepted tokens are shown in bold. After each iteration, the accepted token is removed from the string's prefix and added to the final tokenization. A greedy tokenizer that was not data driven would select "then oodles" instead of the more popular phrase "the noodles."

Given a string `s`, we wish to find a reasonable tokenization of `s` based on a corpus of language. We assume that we have processed the corpus into two data structures: a trie,[13] and a *token table*. We will explain how we produced these data structures later.

The operation of the algorithm is shown in Table 5.5. First, we find the longest prefix of `s` for which we have a tokenization. We do this by looking up `s` in the trie and taking the longest match. Next, we find the most frequent tokenization of this prefix using the token table, which gives us a broken version of the prefix, i.e., it splits the prefix into words. We *accept* the first word of the broken prefix as part of our tokenization of `s`. We then repeat this process on the remainder of `s`, each time taking the first word of the most frequent tokenization, until `s` is empty.

Building the trie and token table

We build the token table first. We remove all non-alphabetic n-grams from the corpus, then downcase and aggregate duplicates from the remaining n-grams. We assume that words in passwords may not have the same case as they do in the corpus, so we ignore case when tokenizing. For each n-gram, we create a record containing its unbroken version (removing spaces between tokens), its broken version, and its frequency in the corpus. We collect and sort these records, and call the resulting table a *token table*. The token table is sorted by unbroken strings, which are used as keys to find records in the table. Thus, for any unbroken text in the corpus, we can look up the text in our token table and examine the frequencies of all tokenizations. Our token table contains 1.35 billion rows.

However, it is sometimes necessary to tokenize text which does not appear in the corpus. This is because longer passwords, such as passwords with sixteen characters or more, might contain strings of many tokens. Passwords also might contain random letters suffixed or prefixed to a known n-gram. In both cases, these passwords would go untokenized if our algorithm did nothing more than look up text in the token table. In order to tokenize text which does not appear explicitly in the corpus, we populate a trie with all of the unbroken n-grams from the token table. Tries are optimized for performing prefix searches: given an arbitrary string, we can quickly find the longest prefix of that string in the trie. We use a Python library, marisa-trie, which implements the trie building and lookup operations. It also has a native serialization format so we can keep the trie on disk along with the token table for future lookups.

[13] `https://en.wikipedia.org/wiki/Trie`

5.5.3 Unsupervised tokenization

As described in Section 5.5.1, we can choose to learn untokenized structures from whole passwords, such as `password123!`. However, this does not address password sets which contain common substrings, especially when these substrings contain a mix of character classes. For example, if the substring `p@ssw0rd` appears in many passwords, it would not be learned by any of the tokenization methods discussed so far. Using a Weir PCFG, it would be tokenized by character class into `p`, `@`, `ssw`, `0`, and `rd`, and the linguistic tokenizer does not handle mixed-class strings. `p@ssw0rd` is also a good example of a token which is unique to the password domain and is not a proper word in any language.

Substrings of this form are common when users add symbols to their passwords. In the subset of the RockYou corpus where passwords have minimum length 8 and contain a digit, a symbol, and an uppercase letter, the most common password is `P@ssw0rd`. You also find 74 variations of this including `P@ssw0rd1`, `P@ssw0rd5`, `P@ssw0rd!`, `MyP@ssw0rd!`, `P@ssw0rd11`, etc. We would like to be able to make guesses that fit these patterns, but that do not appear in the training data, such as `P@ssw0rd3`. We could achieve this by tokenizing the above passwords so that `P@ssw0rd` appears as a single token.

Finding common substrings is a nontrivial task that needs to be well defined. If there is no restriction on length, the most common substrings are always single characters, since these will have higher frequencies than any longer substrings that contain them. Imposing a minimum length is not a satisfactory solution either. A longer token such as `password` will never be chosen because `pass` (assuming a minimum length of 4) will always be more frequent, and tokens like `i` and `!` are best left as single characters.

5.5.3.1 Word segmentation

To address this issue, we borrow the algorithm first introduced by Liang and Klein [85] and refined in Berg-Kirkpatrick et al. [3]. The algorithm searches for the most likely "segmentations" (tokenizations) in a corpus of unbroken sentences. The advantage of this algorithm is that it is "unsupervised." In other words, you do not have to provide it with a corpus of language to produce a solution, so it can be applied to languages where we do not know the underlying vocabulary.

Rather than finding common substrings directly, the algorithm tokenizes the inputs using a probability model, where the likelihood of a tokenization is a product of the probabilities of its tokens. Maximizing likelihood, when likelihood is a product of token probabilities, results naturally in a preference for longer tokens. For example, the tokenization [`p@ssw0rd`, `1`] is more likely than [`p`, `@`, `ssw`,

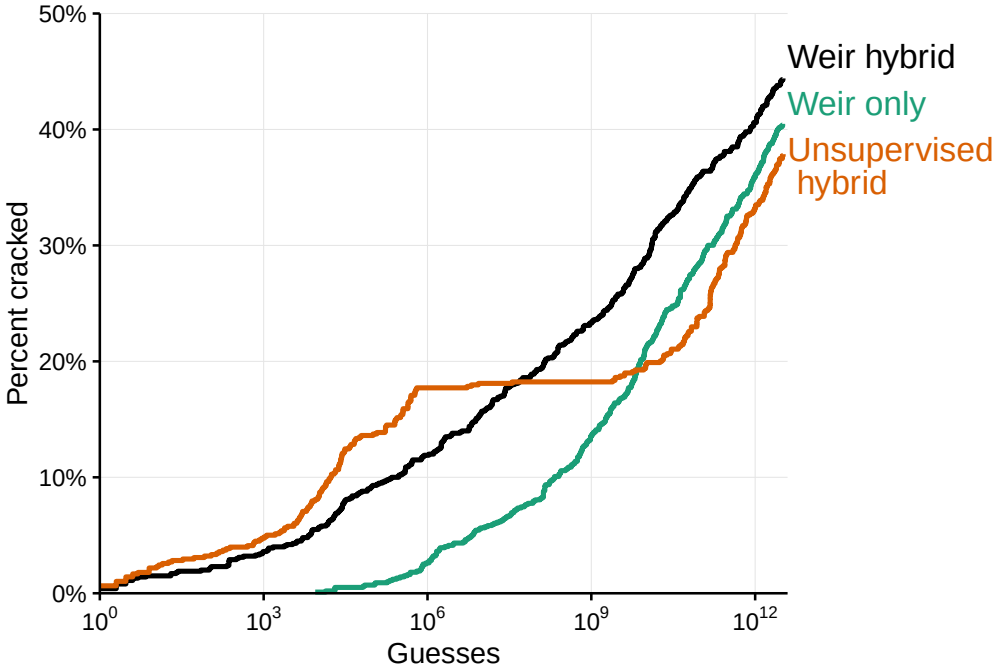


Figure 5.8: Experiment 11 — Guessing curves comparing unsupervised tokenization with hybrid-Weir structures and Weir structures only. The model learned using unsupervised tokenization performs best in early guessing, but this is entirely due to guessing based on untokenized structures. Less than 6% of passwords cracked by the unsupervised-hybrid model were cracked as a result of the unsupervised tokenization approach.

`0`, `rd`, `1`], because the former multiplies two probabilities and the latter multiplies six.[14] This is in spite of the fact that the six tokens have much higher individual probabilities than the two. In fact, this effect is so powerful, that using such an algorithm without alteration actually favors a tokenization where all passwords are single tokens. Therefore, the authors imposed a penalty on token length so that long tokens are exponentially less likely than shorter ones [3]. We made a number of changes to this algorithm to support our passwords use case. We do not describe the algorithm in detail, but we describe these changes in Appendix C.

5.5.3.2 Results

Figure 5.8 shows the result of our evaluation of the word segmentation algorithm on `4class8` passwords. We produced three guessing models for evaluation. The first model uses Weir structures, i.e., we tokenize passwords based on character class. The second model uses hybrid structures, i.e., Weir structures plus

[14]This use of joint probabilities to constrain the number of tokens per password is similar to the hybrid approach from Section 5.5.1 that prefers untokenized structures to tokenized ones.

untokenized structures as described in Section 5.5.1. The third model was built using unsupervised-hybrid structures, i.e., untokenized structures plus structures learned using unsupervised tokenization. We enabled unseen terminal generation and quantized string frequencies for all three models.

Surprisingly, the unsupervised-hybrid model performed even better in early guessing than the Weir-hybrid model which was previously the best model for `4class8`. However, its performance drops off sharply after $\approx$ 600,000 guesses, picking up again after $\approx 10^9$ guesses. Analyzing the guesses made by the unsupervised-hybrid model, we find that its performance is almost entirely owed to the use of untokenized structures. Over 94% of the passwords cracked in this condition were cracked using an untokenized structure, and $\approx$ 600,000 guesses is where the model starts to guess tokenized structures almost exclusively. After 10^9 guesses, the model begins to guess unseen terminals in untokenized structures. In other words, it begins to brute-force entire passwords. The passwords it cracks in this manner are all short passwords.

Conclusion Though the unsupervised-hybrid curve performs well in early guessing, we find that this is entirely owed to untokenized structures and not to the unsupervised tokenization algorithm. We selected parameters for the algorithm that are similar to the original work [3], and found that this did not produce good results. Therefore, we do not recommend the use of our implementation of this algorithm for password modeling.

Instead, we suggest using the Weir-hybrid approach, with a penalty applied to tokenized structures. The difference in probability between the untokenized and tokenized structures in the unsupervised model are greater than in the other models, so the untokenized guesses are made first, which seems to be very beneficial in this case. Applying a penalty should help the Weir-hybrid model achieve similar performance in early guessing to the unsupervised-hybrid model, yet continue to perform well after the untokenized structures have been exhausted.

That said, the unsupervised algorithm has a number of parameters and it might be successful if configured differently. We might also be failing during maximization of the algorithm's likelihood function. The tokenizations produced by the algorithm are based on a complex likelihood model that is optimized over the entire input corpus. Anecdotally, we noticed that tokenizations seemed much better on small datasets than on larger, more realistic ones. We do not know if the fault lies with the likelihood model itself when confronted with a large dataset, or if the algorithm is simply failing to find a good solution with its optimization

method.[15] Issues of this type are common in machine learning, and investigating them is outside the scope of this work.

[15]The optimization method used is L-BFGS [3, 86].

Chapter 6

Analysis of Guessing Data

The result of the guess-calculator framework is a set of *guessing data* for various policies. Guessing data contains a guess number for all cracked passwords in a test set, within some maximum number of guesses that is known (the *guess cutoff*). We can use this data to determine the percentage of passwords that were cracked within the cutoff, as well as for any smaller threshold of guesses.

A "guessing curve" (a term we borrow from Bonneau [6] and introduced in Section 3.2) is built directly from guessing data, and plots the percentage of passwords cracked after some number of guesses, over a range of guesses. Figure 6.1 shows a guessing curve for a simple eight-character policy that we call `basic8`, in which the only requirement was that passwords be at least eight characters long. The guessing data plotted in Figure 6.1 come from 1,000 passwords collected from Mechanical Turk using the procedure described in Section 2.2. We will use this curve as an example for much of the chapter, and show how various statistical metrics can be related to it.

This chapter is split into two main sections. In Section 6.1, we discuss metrics for the guessing data produced by the guess-calculator framework, how guessing curves can provide a more accurate description of guessing data than summary statistics, and statistical tests that can be used with guessing data. In Section 6.2, we present methodological issues that can arise when conducting policy evaluations using the framework, and provide some guidance on dealing with these issues.

6.1 Metrics and statistical techniques for guessing data

Many metrics have been proposed for guessing data, and Bonneau provides a comprehensive survey of these metrics [7]. The most relevant of these are "partial

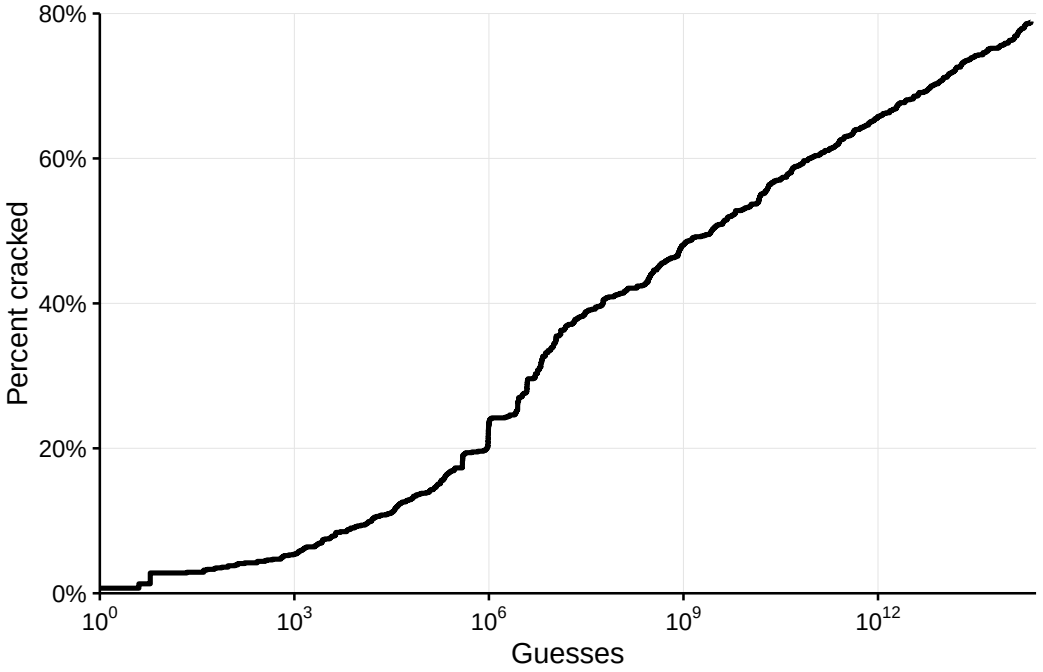


Figure 6.1: Experiment 1 — Guessing curve for the `basic8` policy showing percentage of passwords cracked over a range of guesses.

guessing metrics" where it is assumed that an adversary is willing to exert a fixed amount of effort, i.e., a maximum number of guesses, against each account. We describe these metrics and show how they can be applied to guessing data in this section. Because the guess-calculator framework can only generate guess numbers up to the guess cutoff, partial guessing metrics are most appropriate. The metrics are described in Section 6.1.1 and illustrated using guessing curves. Guidance on comparing policies using the metrics starts in Section 6.1.2. Finally, section 6.1.4 covers survival analysis, a suite of statistical techniques for analyzing guessing data.

Interpretation

Partial guessing metrics were originally intended to be used with true probabilities, not empirically observed ones [7, 13, 117]. When used in this way, the values computed are true values, with no statistical error. Though it is fairly easy to compute these metrics using guess numbers, we also need to account for error.

In this thesis, we only consider the error that comes from having a *sample* of guess numbers as opposed to having the whole distribution of guess numbers for a particular policy. In statistical parlance, we consider guess numbers to be sampled measurements of an independent variable, where our dependent variables are

the proportion of cracked passwords after some number of guesses, or the rate at which passwords are cracked per guess. The common statistical techniques discussed in this chapter account for this type of error.

However, there are other sources of error that it would be prudent to consider, such as error in the guess numbers themselves. This type of error can arise from many sources: a mismatch between our guessing algorithm and the algorithm of a true adversary, a lack of training data, poorly chosen training data, a weakness in our algorithm, etc. We make an effort to select our training data well, and model techniques that we know are used by professionals [54], but statistical techniques cannot mitigate deficiencies in our model, especially because we do not have data to model the mismatch between our guesses and that of a true adversary. In Chapter 7, we show that the improved guessing model introduced in this thesis is able to model passwords better than previous PCFG models, and thus crack more passwords. This suggests that our models are closer to approximating a more sophisticated adversary, though we are not sure how much further we might have to go.

It is important to keep this in mind when interpreting any password-strength results. As we stated in Section 1.2, we do not have sufficient data to compare the true probability distributions induced by two password policies, and samples from two policies are typically not large enough to compare directly. Instead, we assign guess numbers to passwords and compare guess numbers between samples. These guess numbers are generated by specific guessing models that may or may not reflect the behavior of real-world adversaries.

6.1.1 Partial guessing metrics

There are three partial guessing metrics, as described by Bonneau [7], that we consider in this section. We provide formulas for each metric, and also show how each metric corresponds to a function of a guessing curve. The original formulas Bonneau provides [7] are not directly applicable to our guessing data, because they are based on probabilities from a known, ideal probability distribution and not guess numbers. Luckily, formulas that are based directly on guess numbers are quite simple, as seen below.

β-success-rate ($\lambda_\beta(\mathcal{P})$) measures the expected proportion of accounts cracked for an adversary that makes a fixed number of guesses (β) per account against a given policy ($\mathcal{P}$) [7, 13]. In the original papers, $\mathcal{P}$ is simply a probability distribution. We can substitute a policy for $\mathcal{P}$ here, because a policy induces some probability distribution over passwords.

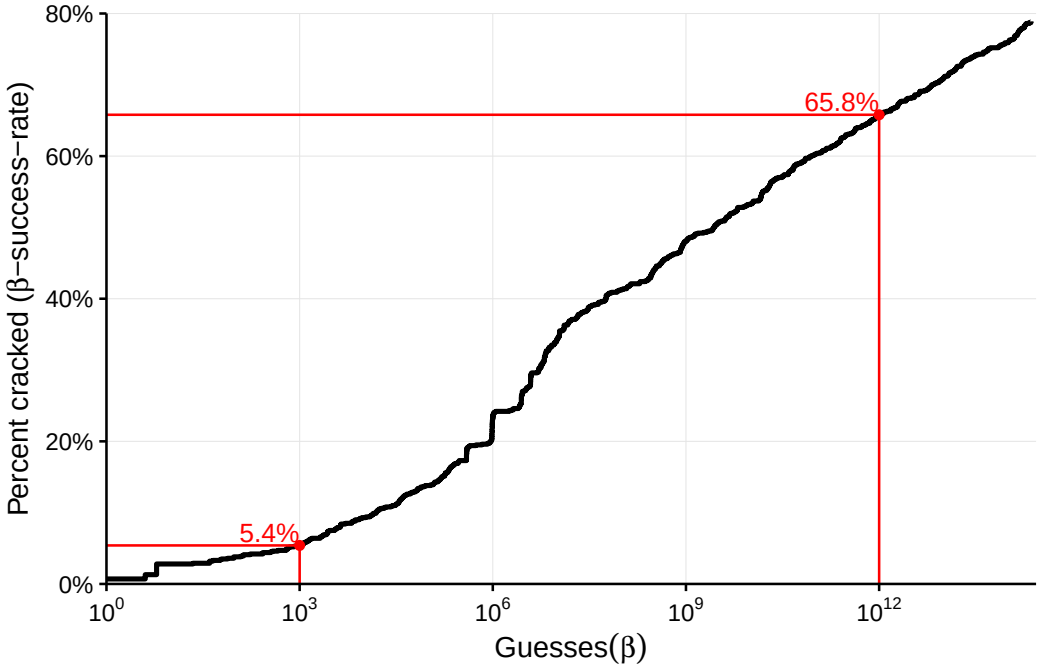


Figure 6.2: Experiment 1 — Guessing curve for the `basic8` policy, annotated for $\beta = 10^3$ and $\beta = 10^{12}$ guesses. The percentages above each point show the estimated β-success-rate for the corresponding number of guesses, based on the test sample.

$$\overline{\lambda}_\beta(\mathcal{P}) = \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}_{g_i \leq \beta} \tag{6.1.1}$$

Equation (6.1.1) presents a formula for estimated β-success-rate ($\overline{\lambda}_\beta$) based on a set of guessing data, where g_i is the guess number assigned by the guess calculator to test password i, N is the number of test passwords, and $\mathbb{1}$ is the indicator function, which is 1 when its condition $g_i \leq \beta$ is true and 0 otherwise. Note that this is exactly the same formula as is used to compute the empirical CDF over guess numbers, so it can be computed easily using standard statistical libraries, such as the "ecdf" function in R.

Recall that a guessing curve plots the percentage of passwords cracked in a sample, over a range of guesses. Therefore, our estimate of β-success-rate is simply the value of the guessing curve at β guesses. For example, as shown in Figure 6.2, the estimated β-success-rate for $\beta = 10^3$ is about 5.4% and for $\beta = 10^{12}$ about 65.8% for the shown policy $\mathcal{P}$. If the curve is vertical at a given β, i.e., at least one account was cracked at this guess, we take the maximum proportion on the curve at the given value.

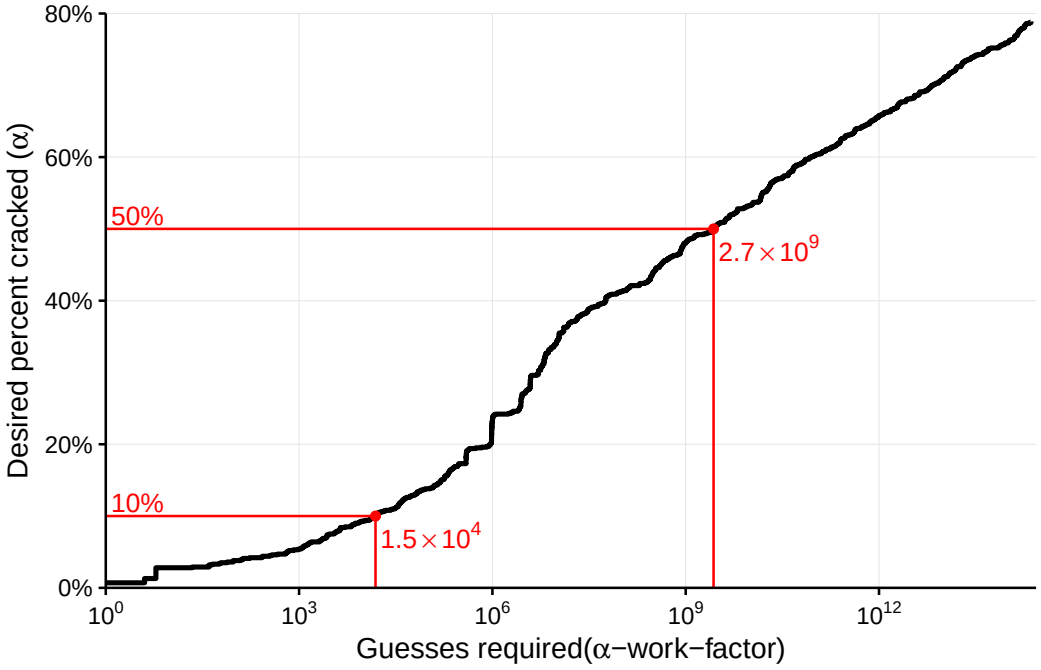


Figure 6.3: Experiment 1 — Guessing curve for the `basic8` policy, annotated for $\alpha = 10\%$ and $\alpha = 50\%$ as the desired percentage cracked. The numbers beside each point show the estimated α-work-factor, or the required number of guesses, based on the test sample.

α-work-factor ($\mu_\alpha(\mathcal{P})$) measures the expected number of guesses needed per account to crack some desired proportion of accounts (α), for a given policy ($\mathcal{P}$) [7,117]. It is the inverse of β-success-rate on a guessing curve.

$$\overline{\mu}_\alpha(\mathcal{P}) = \min_j(\overline{\lambda}_j(\mathcal{P}) \geq \alpha) \tag{6.1.2}$$

Equation (6.1.2) presents a formula for estimated α-work-factor based on finding the inverse of the β-success-rate. Like β-success-rate, it can also be computed easily using standard libraries, such as the "quantile" function in R using "inverse of the empirical cumulative distribution function" mode. As shown in Figure 6.3, the α-work-factor for $\alpha = 10\%$ is $\overline{\mu}_{10\%} \approx 15$ thousand guesses and $\overline{\mu}_{50\%} \approx 2.7$ billion guesses.

α-guesswork ($G_\alpha(\mathcal{P})$) measures the expected number of guesses needed per account to crack some desired proportion of accounts (α), accounting for the fact that an adversary can stop immediately after cracking an account [7]. In other words, it is unnecessary to make the maximum number of guesses per account if an account is cracked earlier. This is a very practical metric, since it measures a value proportional to the average effort that would be expended by an adversary.

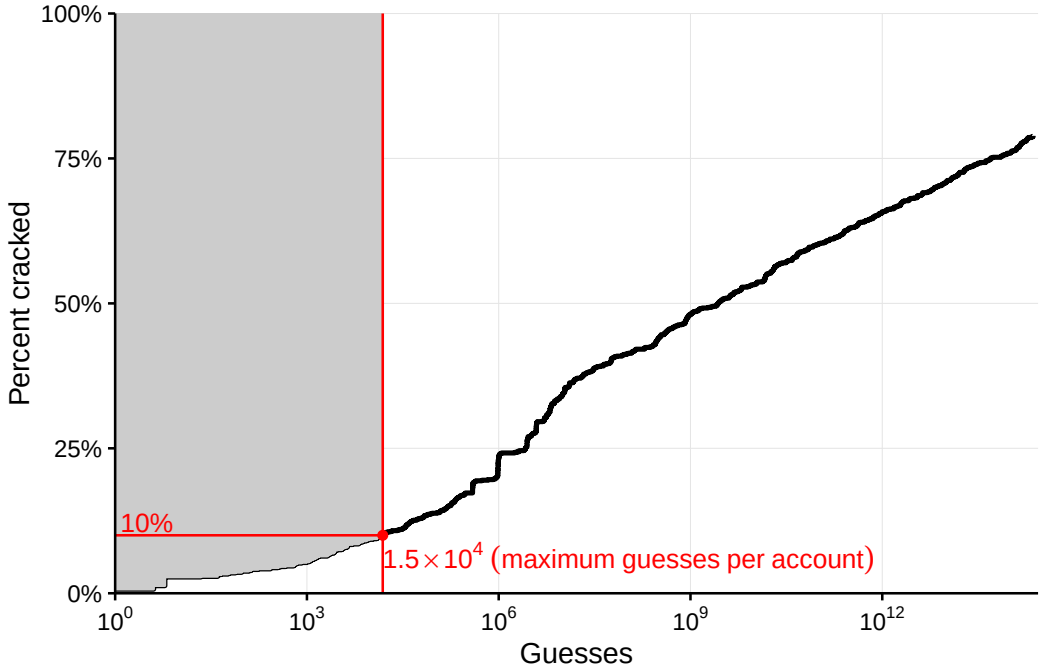
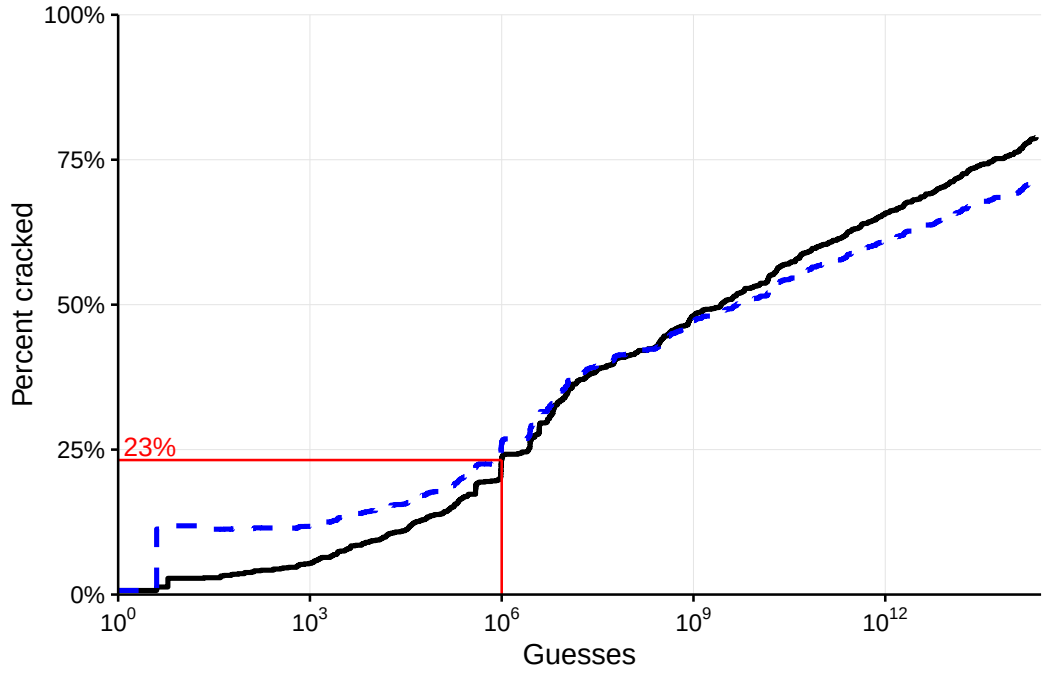


Figure 6.4: Experiment 1 — The area above the guessing curve and up to a maximum number of 15,347 guesses. The total shaded area approximates the α-guesswork needed to crack 10% of passwords. Note that the x-axis is in log-scale, so the area as shown is not the *true* area of the region.

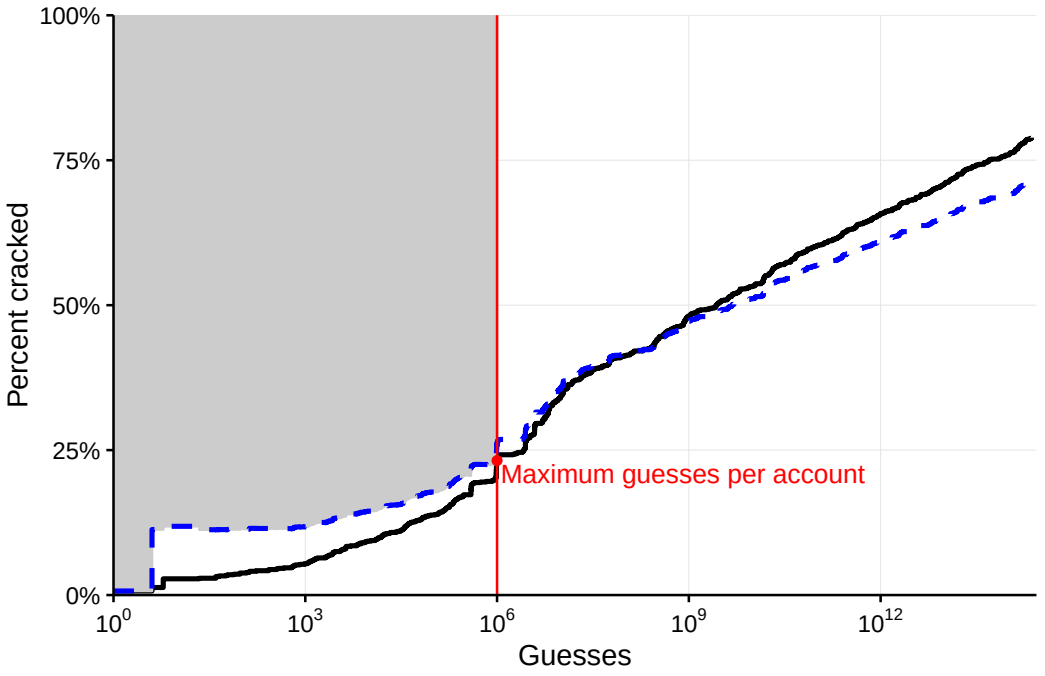
$$\overline{G}_\alpha(\mathcal{P}) = \frac{1}{N}\sum_{i=1}^{N} g(i)\,,\; g(i) = \begin{cases} g_i & \text{if } g_i \le \overline{\mu}_\alpha(\mathcal{P}), \\ \overline{\mu}_\alpha(\mathcal{P}) & \text{if } g_i > \overline{\mu}_\alpha(\mathcal{P}) \text{ or unassigned.} \end{cases} \tag{6.1.3}$$

Equation (6.1.3) presents a formula for an estimate of α-guesswork, based on an estimate of α-work-factor. It cannot be computed using standard libraries, but the computation is fairly easy. The intuition behind this formula starts by summing over the guess numbers up to the α-work-factor. For example, when $y = \mu_{50\%}(\texttt{basic8}) \approx 2.7 \times 10^{12}$, we can sum over all guess numbers up to and including y to obtain the total number of guesses used to crack 50% of accounts. To complete the calculation, we assume the adversary makes $\mu_{50\%}$ guesses per remaining account, quitting when the account is not cracked.

We can visualize this value from the guessing curve, by assuming that the curve represents guessing data from an infinite number of samples. As shown in Figure 6.4, the α-guesswork for $\alpha = 10\%$ can be seen as the area between the y-axis and the guessing curve up to $y = \alpha$ and then the area *above* the curve from α to 100% (to account for all the guesses made on accounts that were not cracked).



(a) Two policies with crossed curves at 10^6 guesses



(b) Visualized guesswork for dashed curve

Figure 6.5: Two policies whose guessing curves cross. At 10^6 guesses, the percentage of the sample that was cracked (also known as the β-success-rate) is the same for both curves. However, the guesswork required for the dashed curve is less than for the solid curve, indicating that the dashed policy is weaker for an adversary restricted to 10^6 guesses. Note that the x-axis is in log-scale, so the areas shown are not the *true* areas of the regions.

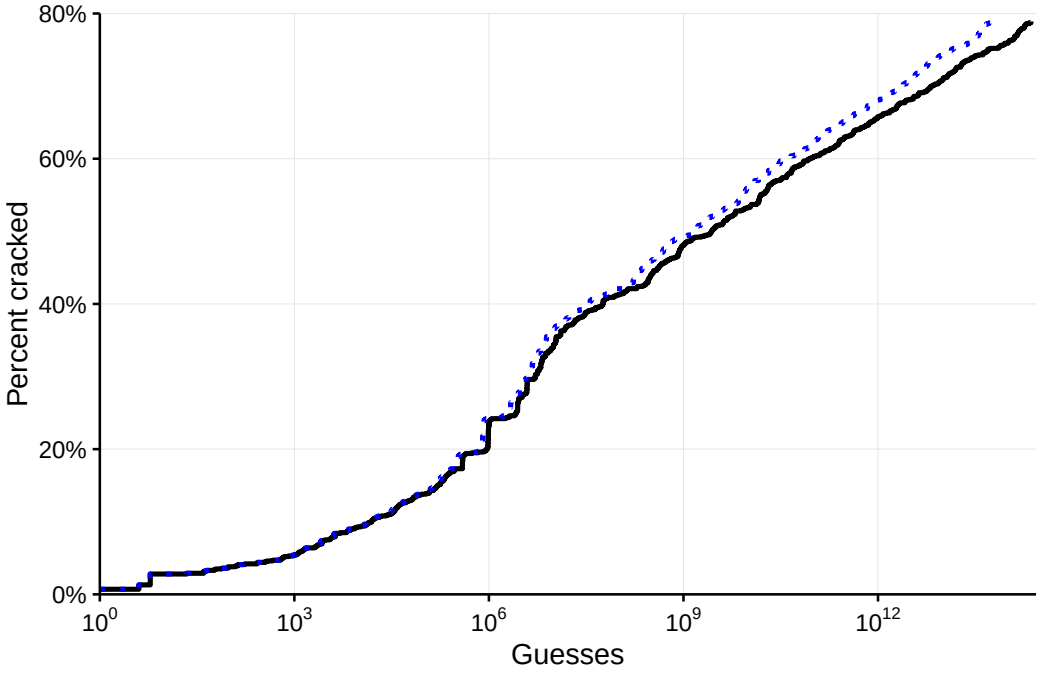


Figure 6.6: Experiment 1 — Guessing curve (β-success-rate, solid black line) compared with guesswork curve (α-guesswork, dotted blue line) for the `basic8` policy.

Figure 6.5(a) shows our `basic8` policy, and a fabricated policy which has equivalent β-success-rate and α-work-factor at 10^6 guesses, but requires less α-guesswork. As seen in Figure 6.5(b), if the α-guesswork of two policies are compared at a point where both policies' guessing curves cross, α-guesswork can identify one of the policies as weaker, since it requires fewer guesses in expectation to reach the same proportion of cracked accounts. This can be seen visually as the top curve having a smaller shaded region than the bottom curve would have.

The α-guesswork for a policy tends to be very close to the α-work-factor for relatively low values of α. Figure 6.6 shows a "guesswork curve" that plots the computed guesswork for a spectrum of α values, using similar axes to our standard guessing curves. Because α-guesswork is a mean score, it is easily skewed by the uncracked accounts at low values of α, which require the adversary to expend a maximum number of guesses. For larger values of α, differences become more pronounced.

Guessing curves

We generally prefer to work with curves rather than summary statistics. The metrics we have discussed all require the specification of a parameter, α or β, and it is usually not apparent which value to choose. As we have shown, a guessing

curve can plot any of the above statistics over a range of α or β values. This allows for policy evaluations based on much more information than a single statistic.

Section 6.1.4 discusses hypothesis tests that can be used to compare policies across entire guessing curves. When guessing curves cross, as shown in Figure 6.5(a), it can be argued that neither curve is strictly better than the other, unless one prefers particular values of α or β. In these cases, one could select specific points at which to compare metrics. This approach is discussed in Section 6.1.2.

6.1.2 Comparing β-success-rates at specific points

In previous work [67,134] our research group focused on the β-success-rate, or the percentage of passwords cracked after some number of guesses, for various values of β and between various policies. For example, we wanted to determine if one policy has significantly more, or fewer, cracked passwords after 10^6, 10^{12}, or the cutoff number of guesses. Comparing the β-success-rates between policies is relatively easy: we can view guessing data for a single policy as a sample from a binomial distribution, where each password is either cracked or not cracked after β guesses. This produces an $n \times 2$ contingency table, on which we can run a χ^2 test (chi-square) to determine if two or more policies are significantly different. If more than two policies are being compared and the overall, or "omnibus," test is significant, we compute pairwise χ^2 tests corrected for multiple comparisons using Holm-Bonferroni correction [60,124].

Fisher's exact test Fisher's exact test can be substituted for the χ^2 test if the cell counts are not too high to prevent Fisher's test from being used. It **should** be substituted if any cell counts are low, as this can cause the χ^2 test to be inaccurate. We have encountered at least one reviewer who disagreed with this method, claiming that since Fisher's test was designed under an assumption of fixed marginals, it is not appropriate to this application.[1] However, it is accepted practice that Fisher's test can be used in these situations [96], and that attempts to construct new tests that relax this assumption are dangerous [35]. In our own testing, we have found that the χ^2 test and Fisher's exact test agree in all cases where we have compared them, but we prefer to use the more exact Fisher's test where possible.

[1]The "fixed marginals" assumption refers to a situation in which the total number of each outcome is fixed, e.g., the total number of cracked and uncracked passwords is known in advance, but we do not know which policy might "choose" more or fewer cracked passwords than the other.

6.1.3 Comparing policies using resampling

Alternatively, we could consider how much work an adversary would require to crack some percentage of passwords, and compare this between policies. In other words, we might want to compare the α-work-factor or α-guesswork between policies. There are no standard statistical tests for this, but relevant procedures exist for computing confidence intervals. Once confidence intervals have been computed, we can say that two policies are significantly different if their confidence intervals are disjoint.

Example: comparing α-guesswork

Let us consider how α-guesswork might be used to compare policies. Recall that in the α-guesswork metric, our threat model is an adversary that makes some maximum amount of guesses per account, but does not waste effort on accounts once they are already cracked. We might want to compare polices based on some threshold of guesswork, e.g., the adversary is willing to make an *average* of 10^6 guesses per account. Given this threshold, we would like to know if the percentage of passwords cracked is significantly higher for one policy compared with another. In other words, we choose an α-guesswork value of interest and want to compare the corresponding αs for each policy.

This is very similar to the scenario in which we compared β-success-rates between policies, with one important difference. When comparing β-success-rates, our β was chosen by us as 10^6 or 10^{12}, independent of the data. However, guesswork is computed in a way that explicitly includes the values of other guess numbers in the sample,[2] so α-guesswork values are not independent observations. This violates the underlying assumptions behind many standard statistical tests, so using them on guesswork measures would be inappropriate. A resampling technique, as described below, could be used instead.

Constructing confidence intervals with resampling

We have previously used the "basic bootstrap" technique [23, 38] to construct confidence intervals for mean password length, mean composition factors (number of digits, symbols, etc.), and estimated entropy values [95]. Though it is easy to construct confidence intervals for mean length using, e.g., standard error, this implicitly assumes a normal distribution. In contrast, password length is heavily left-skewed, since most users will create passwords with minimum length. We can use the bootstrap technique to construct more appropriate intervals.

[2] See Equation (6.1.3) on page 118.

To compare α-guesswork between policies, we could create confidence intervals for α for each policy, using a resampling technique [28]. The resulting confidence intervals can then be compared. This technique (or other resampling techniques) can be applied to many password metrics, though it is computationally intensive. Computing an interval involves creating a number of "replicates" through sampling from the original dataset and computing the metric of interest on the replicate. In our work, we typically compute 1,000 replicates. See Good [53] for a thorough discussion of resampling methods and their use in constructing confidence intervals.

Comparing α-work-factor

Confidence intervals for α-work-factor can be computed without resampling. The α-work-factor estimate for a policy is equivalent to the α quantile of the guessing data. The quantile is computed by ordering the guess numbers, with uncracked passwords assigned a guess number of ∞, and finding the guess number at rank $\lceil N\alpha \rceil$, where N is the size of the sample. There are methods for computing confidence intervals for quantiles that are simple and computationally trivial, such as inverting the binomial confidence interval [34]. Resampling methods can also be used, but they require significantly more computation.

Comparing policies using permutation tests

Another approach to comparing policies is permutation testing. Permutation testing requires an additional assumption of exchangeability but can be more powerful than the bootstrap approach [53]. It is an alternative to constructing confidence intervals. Instead, a metric of interest is used to test the null hypothesis that two policies are drawn from the same distribution. The original procedure involved exchanging data points between two samples, computing the metric of interest on both samples, noting the absolute value of the difference, and repeating for all permutations of the two samples [28]. In practice this is not feasible, so random sampling without replacement from the space of all permutations is used instead. The p-value of a permutation test is the fraction of all differences that meet or exceed the difference between the original two samples.

For example, imagine two samples $\mathcal{A}$ and $\mathcal{B}$ with sample means of 0 and 10, respectively. Take the case where the samples are drawn from populations with true means 0 and 10, respectively. By exchanging data points between the samples, we expect to bring their means closer together, so the fraction of differences greater than 10 will be very low. Conversely, if the true means are the same yet the sample means are still 0 and 10, then we expect that the original distribution (and the

samples) have high variance. Exchanging data points between these samples will sometimes yield new samples that are further than 10 apart. If this happens more than 5% of the time, i.e., $p > .05$ we will not reject the null hypothesis that the two means are the same.

We can execute the same procedure with samples from password policies. We have done this previously using estimated entropy as our metric of interest [76]. To illustrate the procedure, imagine that the estimated entropy of sample $\mathcal{A}$ is $E(\mathcal{A}) = 25.0$ bits, and the estimated entropy of sample $\mathcal{B}$ is $E(\mathcal{B}) = 26.0$ bits. Their difference is 1.0 bits. We then exchange passwords between the samples creating two new "mixed" samples A and B, and compute the entropies of A and B and their absolute difference: $|E(B) - E(A)|$. We repeat this process hundreds of times, careful that we do not construct the same pair of samples more than once. Finally, we calculate a p-value equal to the fraction of absolute differences greater than 1.0 bits.

6.1.4 Survival analysis

To compare the guessability of policies using partial guessing metrics, we must first choose parameters of interest. For example, if we want to compare the β-success-rate between two policies, we might be interested in a strong adversary and choose $\beta = 10^{12}$. We could then use a Fisher's exact test to compare the proportion of passwords cracked after 10^{12} guesses for each policy.

However, it might be unclear what value of β to choose. One could run statistical tests with multiple β values, but this shifts the problem from selecting a single value to selecting a candidate set of β values. Running multiple tests also requires multiple testing correction, which decreases the power of the analysis.

Another approach is to compare guessing curves over the full range of available values. Suppose we have guessing data for two policies `A` and `B` as shown in Figure 6.7. Policy `A` appears to be stronger, but this might not be statistically significant. In other words, the observed difference between the two policies might be due to randomness in our samples, and we would like some confidence that one policy is better than another.

Two tools that we can use, the G^1 test and Cox regression, come from the literature of survival analysis. This family of techniques was designed to discover factors that influence mortality, and works with datasets in which we record times at which fatal events happen to subjects. A nice feature of survival analysis is that

[3] These curves are provided for demonstration and were both generated from the same policy, rather than two different policies. To produce a difference between the curves, one was generated with hybrid structures (Section 5.5.1) and one without.

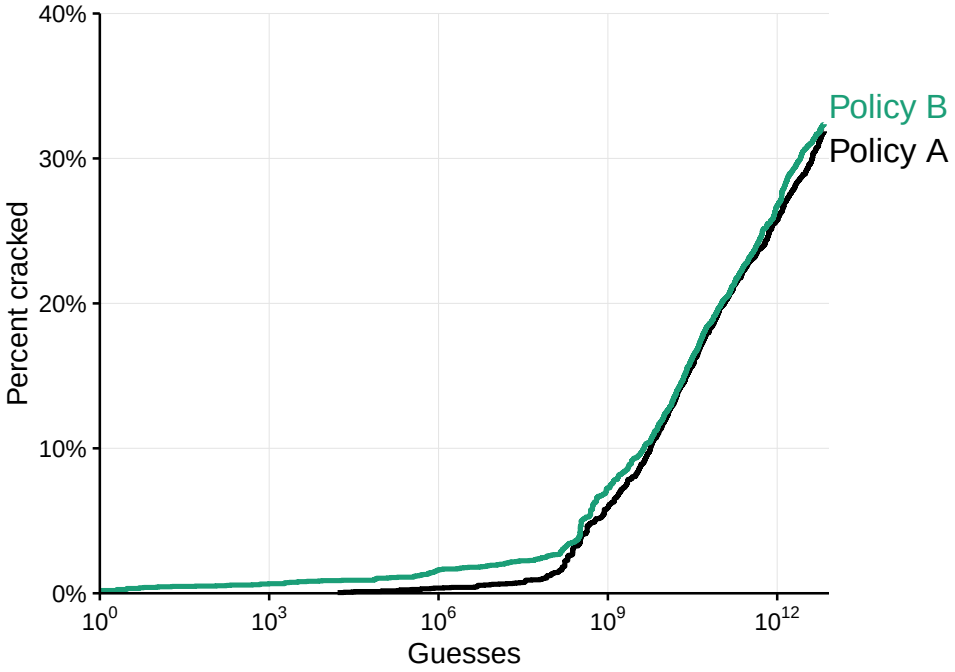


Figure 6.7: Experiment 3C — Example guessing curves for two policies that are close in performance.[3]

it was designed to be used in a setting where a study might end before fatal events are observed. When this occurs, the data notes the time that each subject was last "seen." In the context of passwords, we can consider cracking a password to be a fatal event and mark time using the number of guesses before a password is cracked. The guess cutoff is the censoring point, after which we no longer record observations.

Unlike most of the partial guessing metrics described previously, survival analysis was not designed around the concept of adversary effort. In α-guesswork, we assume that the adversary will reallocate resources after cracking a password so that energy is not wasted on guesses after that point. The survival analysis techniques described here are not compatible with such an assumption. They are most similar to examining the β-success-rate over a large range of guess numbers.

G^1 test

The Wilcoxon test, also known as a Mann-Whitney U test, is a popular nonparametric test of whether one sample is significantly greater than another [59, 90, 149]. A modification of this test for survival analysis exists under various

names: the Peto-Peto generalization of the Wilcoxon test [114] and the G^1 test.[4] For simplicity, we refer to this test using the latter term.

The G^1 test can be used to compare two sets of guessing data under a null hypothesis that both data sets were drawn from the same distribution. It has the additional property of weighting differences in early parts of the curve more heavily than later parts. This is an important feature of the test. Consider a sample of 100 passwords. If 50% of these passwords are cracked by time t, there are only 50 passwords left on which an event might occur. In other words, for times greater than t, our sample size is effectively cut in half. The G^1 test accounts for this and has less statistical power for differences in later parts of the curve than earlier parts.[5]

The G^1 test computes $p = 0.28$ on the two samples in Figure 6.7. This means that we cannot reject the null hypothesis that both samples were drawn from the same distribution. In other words, we cannot say that the two policies we tested are different. Consider what we get by comparing β-success-rates at specific points: χ^2 tests are significant for $\beta \in \{10^3, 10^6, 10^9\}$ but not for $\beta = 10^{12}$, which creates confusion about whether the two curves are significantly different. A comparison of values across the available range of βs using a G^1 test gives us a single p-value with which to interpret our results. If we are not at all interested in events beyond some guess number, say 10^4, we could modify our guessing data so that only password cracks before $\beta \leq 10^4$ are counted as "fatal" events. This allows us to get the benefit of comparing curves over a range of values with a single test, while restricting that range only to values of interest.

The computation of the G^1 test also provides us with scores for each policy that tell us the direction of any observed effect. Based on a null hypothesis that both curves are drawn from the same distribution, the test computes an expected number of passwords cracked for each policy. We can then examine the difference between the observed and expected number of cracks for the two policies. One policy will have an observed count greater than expected, while the other policy will have less. The policy with fewer observed cracks than expected is observed to be stronger, and the other policy is observed to be weaker.

Cox regression

A similar approach to the G^1 test is Cox regression. Unlike the G^1 test, which is nonparametric, Cox regression is considered "semi-parametric" [147]. It treats each

[4] This name comes from a family of tests known as G^ρ tests where setting $\rho = 1$ yields the Peto-Peto test [55].

[5] Note the terms *early* and *late* as used here refer to the proportion of passwords remaining in each policy and not time.

Factor	Coef.	Exp(coef)	SE	p-value
Policy B	0.0361	1.04	0.0384	0.35

Table 6.1: Cox regression results for the two guessing curves of Figure 6.7. Policy B is 1.04 times easier to crack than Policy A, but this difference is not significant.

guessing curve as a function of a nonparametric "arbitrary and unknown function of time" [36] and independent variables which are represented by regression coefficients. This means that the guessing curves of Figure 6.7 can take any shape and the Cox regression will only look at differences between the curves.

We explain Cox regression in more detail using the guessing curves of Figure 6.7. The result of a Cox regression on these curves is shown in Table 6.1. To run the regression, we take the policy as a categorical variable with two values, A and B, which is dummy coded with Policy A as the 0 value, also called the baseline. Table 6.1 shows the effect of the policy on cracking. Based on the data, the regression estimates that Policy B is 1.04 times easier to crack than Policy A. In other words, over a given interval, e.g. 1,000 guesses, Policy B will have 1.04 times as many cracked passwords as Policy A. However, this is not statistically significant given the data ($p = 0.35$).

Like other regression methods, we can use Cox regression to analyze results from many experiments at once. For example, we could collect guessing data from four policies: `basic8`, `basic12`, `4class8`, and `4class12`. This is known as a 2-by-2, full-factorial design, where the factors are length and class. Using Cox regression, we could analyze all four experiments at once and estimate the individual effects of length and class on password strength. Another application of Cox regression is in comparing several guessing curves to a single baseline curve at once using a single test.

Differences between methods

Reducing differences down to a single coefficient can oversimplify the relationship between two policies. Unlike the G^1 test, Cox regression estimates a parameter (a regression coefficient) that represents the average difference between policies. This coefficient represents a constant effect which might not represent any real behavior of the guessing curves. Using the example policies A and B from Figure 6.7, there might not be any region of the curves where Policy B has 1.04 times as many passwords cracked as Policy A. It is even possible that Policy B has far fewer passwords cracked than Policy A over the majority of guesses, so long as there are

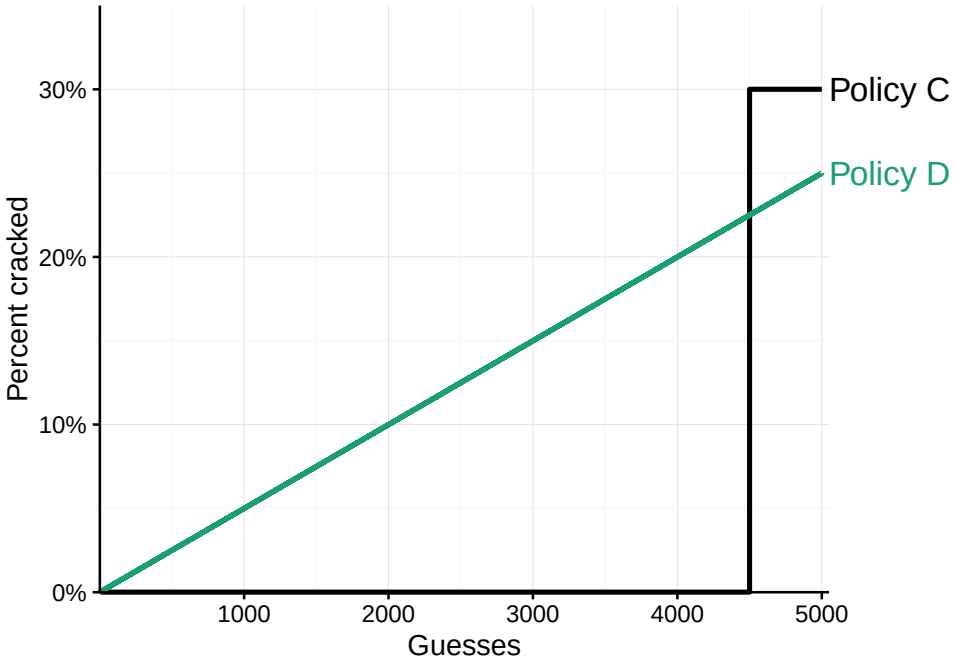


Figure 6.8: Example guessing curves for two policies where Cox regression and the G^1 test produce contradictory results.

some regions where Policy B is bad enough to be estimated as worse overall. In these cases, the result of the G^1 test can differ from that of Cox regression.

Figure 6.8 shows two hypothetical guessing curves where Policy D is worse than Policy C over the majority of the curve, but Policy C is considered worse under Cox regression. The sudden jump in cracks at guess number 4,500 causes Cox regression to estimate that policy C is 1.09 times easier to crack than policy D ($p < 0.001$). In contrast, the G^1 test produces the opposite result: policy D is weaker than policy C ($p = 0.028$). This is a consequence of the G^1 test weighting early cracks more than later ones.

It should not be surprising that different methods produce different results for Figure 6.8. Whenever guessing curves do not strictly dominate one another, it is unclear which curve should be recommended. In these cases, statistical methods can still be used to test for significant differences, but it is up to the researcher to interpret these results carefully before making a recommendation.

6.2 Conducting policy evaluations

As explained in Chapter 4, generating guess numbers for a given test set is mostly an automated process with the guess-calculator framework. However, there are

still many decisions that need to be made by the policy researcher: training data still needs to be selected and weighed, and proxy test sets might be required if passwords created under the target policy are not available. Our goal in this section is to cover some of the methodological issues that might arise when conducting policy evaluations.

6.2.1 Additional training samples

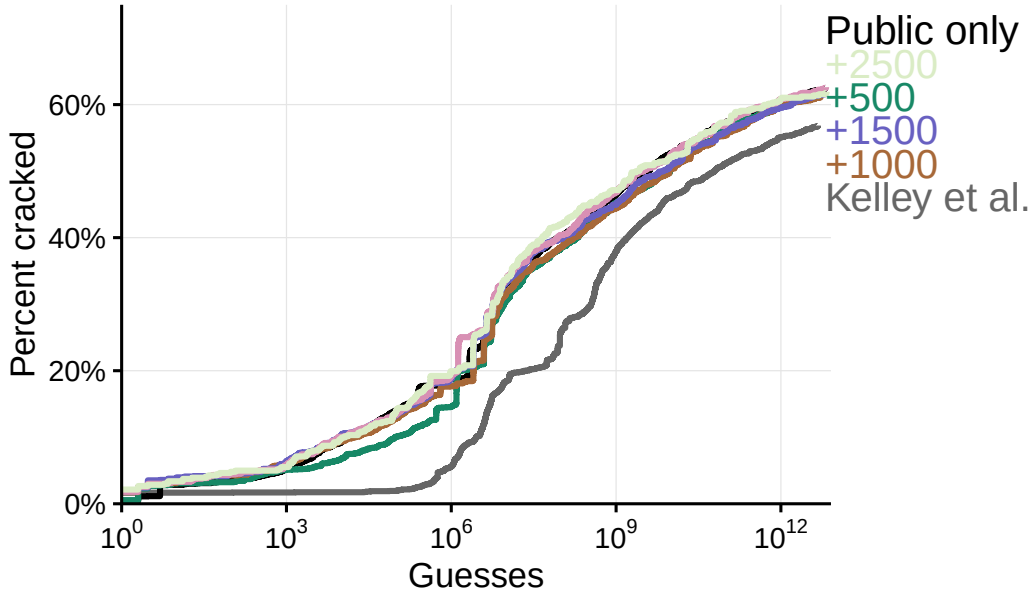
In our previous work (Kelley et al. [67]), we found that additional samples made a significant difference when training a complex policy, like `comp8`,[6] but not when training a simple policy, like `basic8`. With the improvements to the guess-calculator framework made in this thesis, this is no longer the case. Given a large and diverse set of public passwords, such as the RockYou dataset, we can achieve results with public data close to those with additional samples.

Figure 6.9 shows the performance of models trained with only public data compared with models trained with additional samples from the target policy. For each policy, we plot a guessing curve using public data including RockYou (shown in black) plus 500 to 2,500 additional samples in increments of 500. We also show our previous finding from Kelley et al. (shown in grey), in which we trained on the Openwall [39] and MySpace [125] datasets.[7] The Openwall dataset is "a huge list of all the common passwords and words from all the languages with word mangling rules applied" [109]. It has far fewer `comp8` structures than RockYou, which was previously a useful trait—the released implementation of Weir et al. [143] was unable to store all of the structures learned from RockYou in memory, and the Openwall dataset was an available source of `comp8` passwords.

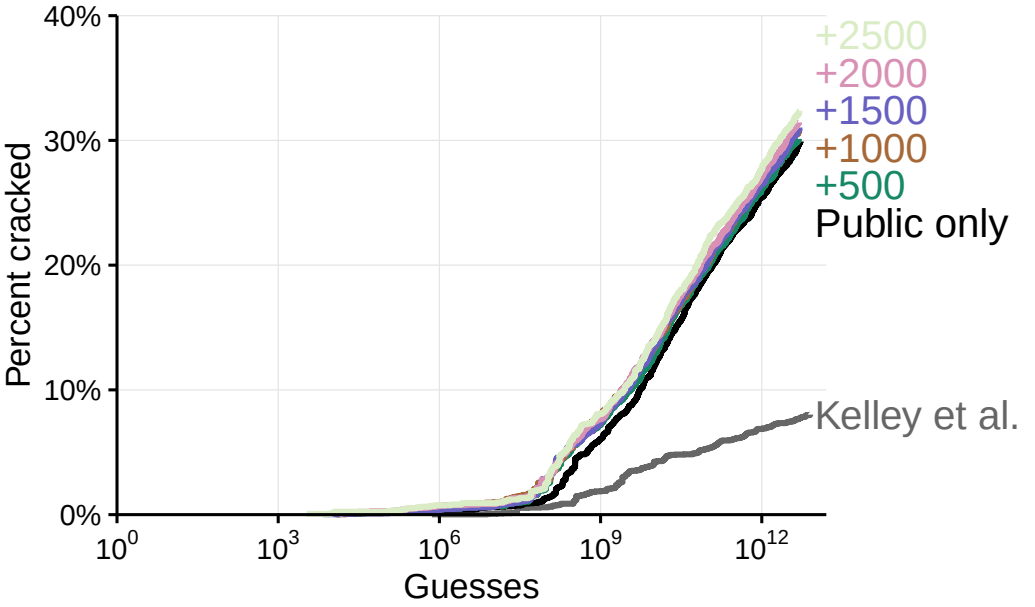
For `basic8`, additional samples from the target policy do not seem to help at all with guessing. Compare this with `comp8`, where each addition of 500 passwords increases the percent guessed by about 0.5% at the cutoff, though this is unlikely to be statistically significant. This suggests that the RockYou dataset is a fine source of data for `basic8`, but does not have enough data for `comp8`. Our results also indicate that the training data used in Kelley et al. [67] was a poor source of data for `comp8` but was reasonable for `basic8`. More details on this dataset are provided in Appendix B.4.4.

[6] Our convention for naming policies is described in Section 2.4.4.

[7] Statistics on these datasets can be found in Sections B.4.3 and B.4.4.



(a) basic8



(b) comp8

Figure 6.9: Experiment 5A (top) & Experiment 5B (bottom) — With the improved guess-calculator framework, a guessing model produced entirely from public data is competitive with one given 2,500 additional samples. The condition labels on the right of the graph are ordered based on percentage cracked by the guess cutoff.

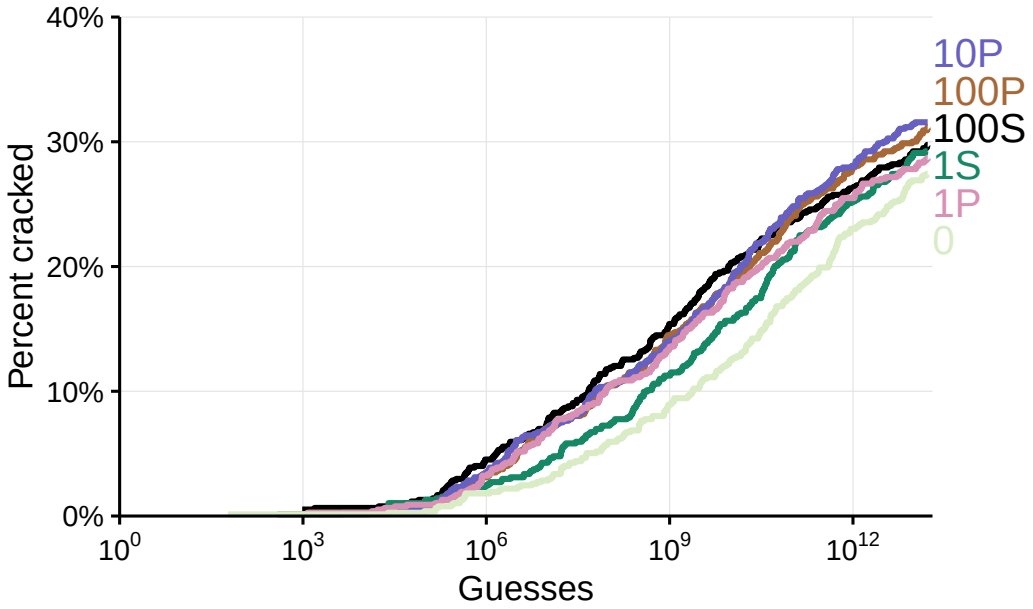


Figure 6.10: Experiment 6 — Guessing curves for six conditions where we manipulated the weight of password samples from the target policy (`3class12`). We find that weighting has little effect on guessing performance in this case, though weighting is generally better than building models with only public data (the `0` curve). See the text for an explanation of the weighting schemes we used.

6.2.2 Weighting training data

In the previous section, samples were scaled so that their cumulative probability was equal to the cumulative probability of other training data. In Section 4.2.1 we called this `P`-weighting. We also introduced the concept of `S`-weighting, which is a more straightforward weighting scheme. In this section, we examine the impact of different `S` and `P`-weighting schemes. We evaluated a baseline scheme with no sampled data, labeled `0`. We also evaluated two `S`-weighting configurations with weights 1 and 100, labeled `1S` and `100S` respectively. Finally, we evaluated three `P`-weighting configurations: `1P`, `10P`, and `100P`. In this particular evaluation, `1P` was approximately equal to an `S`-weighting with weight 105.7, due to the size of the input datasets. We did not measure this before performing the evaluation.

We performed this evaluation using the `3class12` target policy. We selected this policy because it is novel enough that public data would not be an adequate source of training data. Samples were collected from Mechanical Turk as described in Section 2.2 and combined with data from public sources as listed in Appendix B.13.

Guessing curves are presented in Figure 6.10 and Cox regression results are shown in Table 6.2. Though the `10P` model results in significantly more passwords cracked than the baseline `0` model, evidence in favor of the `10P` weighting

Factor	Coef.	Exp(coef)	SE	p-value
100P	0.1821	1.20	0.0942	0.053
100S	0.1409	1.15	0.0952	0.140
10P	0.1999	1.22	0.0939	**0.033**
1P	0.0887	1.09	0.0960	0.360
1S	0.0906	1.09	0.0956	0.340

Table 6.2: Experiment 6 — Cox regression results comparing the guessing curves of Figure 6.10 with public data only ("0") as the baseline. The exponential of the coefficient (Exp(coef)) indicates how strongly that factor being true affects the probability of cracking compared to the baseline category. For example, the 10P scheme cracks 22% more passwords than the 0 scheme, on average. Though this is the only scheme that is significantly better for $\alpha = 0.05$, other weighting schemes are similar in performance.

scheme over other weighting schemes is not very compelling. An examination of Figure 6.10 finds that many of the schemes have curves that cross, so there is no single weighting scheme that is clearly better than all others.

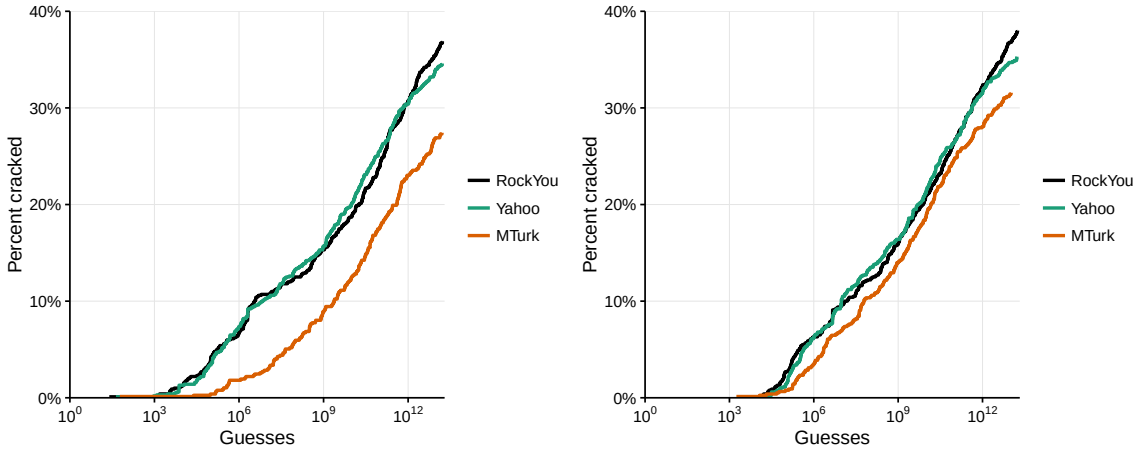
However, the 0 scheme is clearly worse than all others. This suggests that for unusual policies like 3class12, models should be built using collected samples, as this seems generally better than building models with only public data. However, we do not find that particular non-zero weightings have very much effect on guessing performance.

This experiment was limited to a single policy and a small set of weightings, so these findings should be approached cautiously. In Section 7.1, we find favorable results with the 10P weighting scheme, though we did not try other weightings in that evaluation.

6.2.3 Comparing subsets

We also examine the performance of passwords collected under the 3class12 policy to passwords found in public data that meet the 3class12 requirements. This examination is important for those studying complex policies who lack a source of samples from the target. Weir et al. use a subset of passwords from public data as a proxy in their evaluation of strict policies [145]. Note that the public data sources used by Weir et al., and that we use here, consist of passwords collected under much simpler policies: a five or six-character minimum length with no other requirements.

We use the RockYou and Yahoo datasets as inputs to the guess-calculator framework for training. We configure the framework to hold out 1,000 3class12 passwords from each of these sets for testing and use the remainder for training.



(a) Model trained on only public data

(b) Model with `10P` weighting

Figure 6.11: Experiment 7A (left) & Experiment 7B (right) — Comparison of subsets from the RockYou and Yahoo datasets to samples collected from Mechanical Turk (MTurk) for the `3class12` policy. Models were trained on the RockYou and Yahoo datasets, with the `10P` model (right) trained on additional samples from MTurk using `10P` weighting (as described in Section 6.2.2).

We evaluate two models, one built with only public data and one built with MTurk passwords in the training data. Mturk samples are weighted using the `10P` weighting scheme introduced in Section 6.2.2, so we call this the `10P` model. In this model, we used 2,000 passwords for training and test on the remaining 773 passwords. As usual, details on how these models were trained can be found in Appendix B.

Figure 6.11 shows the results of our examination. For a model trained only on public data, we find that passwords created under the `3class12` policy are significantly stronger than passwords found in a `3class12` subset of public data (Cox regression, $p < 0.001$). With the `10P` model however, the gap between the MTurk and other test sets becomes smaller. The RockYou guessing curve is significantly weaker than the MTurk curve ($p = 0.008$) but the Yahoo curve is not significantly different. Therefore, we make no claim that subsets of public data are weaker or stronger than collected passwords. Instead, it is possible that we simply do not have enough training samples to guess MTurk passwords as efficiently as those from large, public datasets.

Discussion This result challenges our findings from previous work. In Kelley et al. [67], we found that subsets of passwords from weaker policies were significantly stronger than passwords collected under a target policy. In an experiment

described in Section 7.4, and previously published in Mazurek et al. [95], we find a similar result: subsets of passwords from leaked datasets can sometimes be significantly stronger than passwords from a target policy. However, we also found that the RockYou set did not seem to exhibit this property. There are at least three methodological differences between this evaluation and previous studies.

First, we use the `3class12` policy, because it can be modeled perfectly by a Weir PCFG. In contrast, previous studies looked at the `comp8` policy that includes a dictionary check, among other requirements. A Weir PCFG makes a large number of noncompliant guesses against such a policy, because it has no way of only generating guesses that pass a dictionary check. This creates inefficiency in cracking the `comp8` policy that might affect different datasets to varying degrees.

The second difference between this and prior studies is the use of improvements to the guess-calculator framework. As we have shown in Chapter 5, the improvements we introduced to the guess-calculator framework can crack policies much more effectively than previous work, and this might also have a different impact on different datasets.

The third difference is in training data. Prior studies included the Openwall dataset [109] in their training data, but we found in Section 6.2.1 that this is a poor data source, at least for the `comp8` policy. Use of this dataset might have created a bias in our results that does not accurately reflect true differences between datasets.

We are unable to draw conclusions about subsets of public data. We find that these subsets can be significantly weaker than passwords collected from Mechanical Turk, but our results also suggest that significant differences might be due to a lack of training data suited to the target policy. This is indicated by the fact that a model trained with 2,000 samples from the target policy cracks our Mechanical Turk test set almost as well as subsets of public data. With even more samples from Mechanical Turk, it is possible that the trend from Figure 6.11(a) to Figure 6.11(b) would continue and the curves would converge. It is even possible that the MTurk curve would overtake the other curves and prove to be weaker than subsets of public data.

Even though we have studied a different policy here than in previous work, our current results challenge the general hypothesis that passwords in the field, which we proxy with passwords from Mechanical Turk, are weaker than subsets of public data. One point in favor of our current results is that their explanation is relatively simple: a lack of training data for cracking Mechanical Turk passwords. In contrast, the explanations for previous results rely on complex user behaviors such as users making weaker passwords for online studies, or that the subset of users who make

complex passwords under a simple policy use a stronger algorithm to generate passwords than the general population. Further research is needed to investigate these explanations in depth, and to improve our ability to address these factors when collecting test data so that we can evaluate policies more accurately.

Chapter 7

Case Studies

This chapter presents several case studies in which the guess-calculator framework and the statistical analysis techniques of Chapter 6 are used to evaluate password policies.

In Section 7.1, we use leaked passwords to evaluate policies from a U.S.-based service and Chinese service. In Section 7.2, we look at mixed-class passwords that are a traditional weakness of PCFG modeling. We find that our improvements to the guess-calculator framework greatly improve our models of these passwords. In Section 7.3, we look at policies involving long passwords, which we have previously found to be strong [67] but might just be difficult targets for our guessing models. We find that our improvements to the framework greatly improve our ability to crack these passwords, though they are still much stronger than shorter password policies.

Finally, in Section 7.4, we describe our use of an older version of the guess-calculator framework in the field to evaluate passwords from Carnegie Mellon University, compare them to various leaked password sets, and use survival analysis techniques to identify factors correlated with password weakness.

The `Public` set We provide the same set of input datasets to the guess-calculator framework to train many of our models. Because structures are pruned as appropriate for the target policy, we can train multiple models with the same input yet produce different PCFGs for each policy.[1] We call this collection of input data the `Public` set, and it is used in many of our evaluations. It is comprised of the RockYou [139] and Yahoo! Voices [31] password sets, supplemented with alphabetic strings from the public-domain Webster's 1934 dictionary [107], the Automatically Generated Inflection Database [1], and unigrams (single words)

[1] Please refer to Figure 4.2 on page 50 which describes how input datasets are processed and used by the framework.

from the Google Web Corpus [15]. We provide details on all experiments in this chapter in Appendix B and configuration files for each experiment can be downloaded from `https://cups.cs.cmu.edu/chili/projects/passwords-gcf-thesis-configs/wiki/Wiki`.

7.1 Analyzing leaked password sets

In this section, we compare the strength of leaked passwords from a U.S.-based service (Yahoo!) to those from a Chinese service (CSDN). Unlike most analyses in this chapter, this analysis uses only leaked datasets and does not utilize the `Public` set. A similar analysis was performed by Ma et al. [88] and it is hoped that this analysis can be easily reproduced by other password researchers using other password cracking algorithms. It should be noted, however, that leaked datasets are not necessarily good training data for one another if their policies are not similar. This issue is discussed below.

The datasets examined in this set of experiments were RockYou, Yahoo! Voices, and CSDN. Statistics on these datasets can be found in Appendix B.4.3.

7.1.1 Target policy

We trained the PCFGs for these experiments so that only six-character or longer guesses were made. We call this a `basic6` policy. This is in contrast to Ma et al. who enforce a minimum length of four except in one experiment [88].

Selecting a target policy for leaked password data is not straightforward. Leaked password sets can combine data from multiple policies, and often have strings that are too short to comply with any realistic policy. As shown in Figure 7.1, the CSDN dataset has a range of string lengths with a small peak at six characters, yet the plurality of passwords are eight characters. There are a few potential explanations for this: noise in the data; password reuse, i.e., users reusing passwords from other websites with more strict policies; or the addition of an eight-character minimum requirement at some point after some users had already created accounts. RockYou had a five-character minimum policy, but the majority of passwords in the dataset are six characters, and eight-character passwords are more frequent than seven-character passwords. The Yahoo! dataset seems to conform to a six-character minimum policy, but, like CSDN, the plurality of passwords are eight characters.

These differences between password sets illustrate that training on one set, such as RockYou, might not be ideal when the target policy is another set, such as CSDN. A model based on RockYou will guess many more six and seven-character

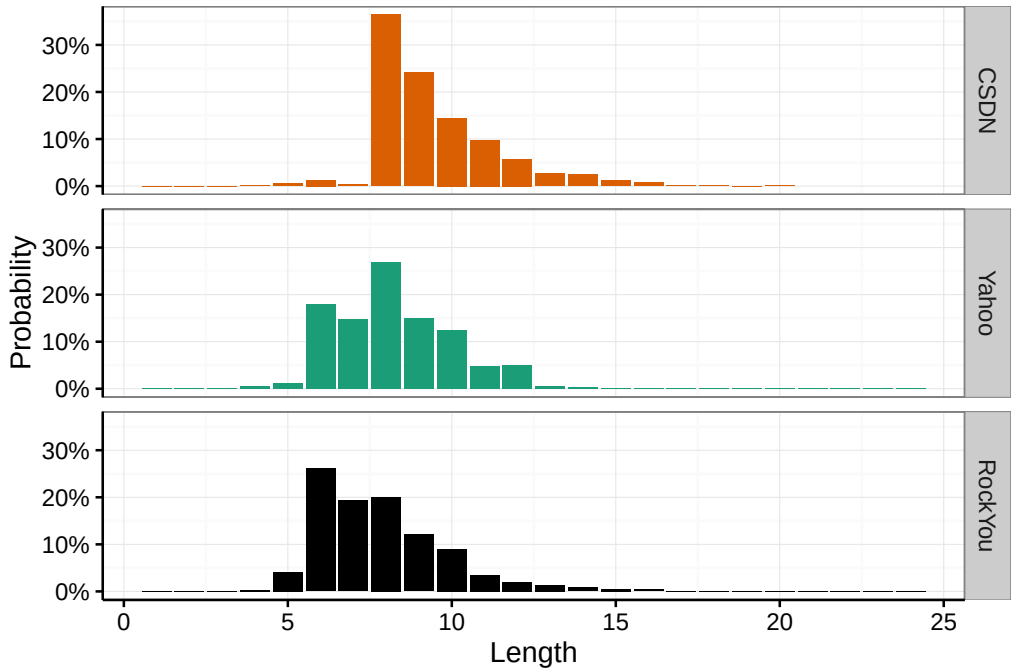


Figure 7.1: Distribution of password lengths across leaked datasets

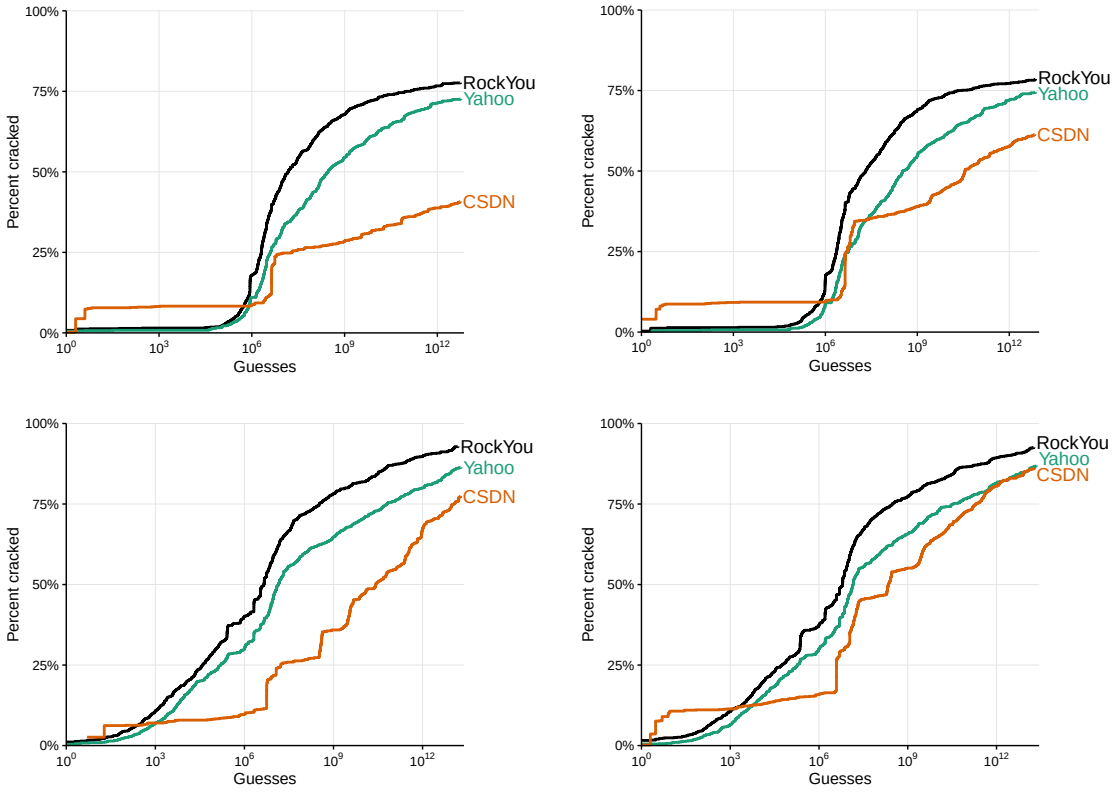
passwords that it should when CSDN is the target. A more in-depth examination of the two sets would find other differences, such as many more passwords consisting solely of digits in CSDN. The work of Bonneau and Xu suggests that CSDN users were constrained by the lack of platform support for characters in their native language, which caused many of them to pick passwords that are all digits [12]. Li et al. found that other Chinese datasets have the same property, and specialized their training data to include generated date and digit strings [84]. From an adversary's perspective, this is an attractive policy to attack.

In this section, we will show how different models perform on these datasets, and how the improvements to the framework introduced in this thesis can produce better models. In particular, we show how a small amount of weighted training data can be used by the guess-calculator framework to produce a model tuned for the CSDN policy.

7.1.2 Results

The four experiments in Figure 7.2 demonstrate the effects of training data and framework improvements on models. The left column is trained only on the RockYou dataset, using `basic6`-compliant passwords for structures and all

(a) Weir 2009 model trained only on RockYou (b) Weir 2009 model trained on all three datasets



(c) Improved model trained only on RockYou (d) Improved model trained on all three datasets

Figure 7.2: Experiment 8A (top left), Experiment 8B (top right), Experiment 8C (bottom left), & Experiment 8D (bottom right) — An evaluation of RockYou, Yahoo!, and CSDN passwords using various models. The top row (*Weir 2009*) shows models trained using settings akin to Weir's original approach for learning a PCFG [146], while the *Improved* model in the bottom row incorporates improvements from this thesis. The left column is trained only on RockYou, while the right column is trained on all three datasets. Test data consisted of 1,000 randomly selected held-out passwords from each dataset.

passwords for terminals. The right column adds Yahoo! and CSDN passwords to the training data. No other data sources were used in this experiment.

Figures 7.2(a) and 7.2(b) contain *Weir 2009* models, which were learned using a methodology similar to that of Weir et al. [146]. To produce the Weir 2009 curves, we ran the guess-calculator framework without unseen terminal generation, using only a character-class tokenizer. In addition, we modify the framework so that alphabetic strings are all assigned the same frequency, as in Weir's original approach [146]. We use the guess-calculator framework, instead of Weir's original implementation, because it allows us to quickly reach trillions of guesses. The *Improved* model in Figures 7.2(c) and 7.2(d) incorporate improvements from this

	10^0	10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8	10^9	10^{10}	10^{11}	10^{12}
Weir 2009 model trained on all three datasets													
CSDN	4.0	8.7	8.7	9.2	9.3	9.3	9.5	34.4	36.2	39.0	45.0	52.7	57.6
RockYou	0.3	1.4	1.4	1.4	1.5	2.5	16.7	45.3	59.1	68.9	74.0	75.9	77.2
Yahoo	0.0	0.5	0.6	0.7	0.7	1.2	8.6	28.7	41.8	54.7	61.8	67.2	72.0
Improved model trained on all three datasets													
CSDN	0.2	10.7	11.1	11.4	12.7	14.5	15.9	31.3	46.5	55.1	64.8	72.7	80.5
RockYou	1.6	2.4	4.5	10.5	18.6	27.3	37.2	57.4	71.9	77.3	82.1	86.6	89.3
Yahoo	0.5	0.8	2.5	6.5	14.5	23.0	30.1	46.0	59.2	65.7	72.4	76.9	81.5

Table 7.1: Experiment 8B (top) & Experiment 8D (bottom) — Percent of sample cracked by Weir 2009 and Improved models trained on all three datasets after a given number of guesses.

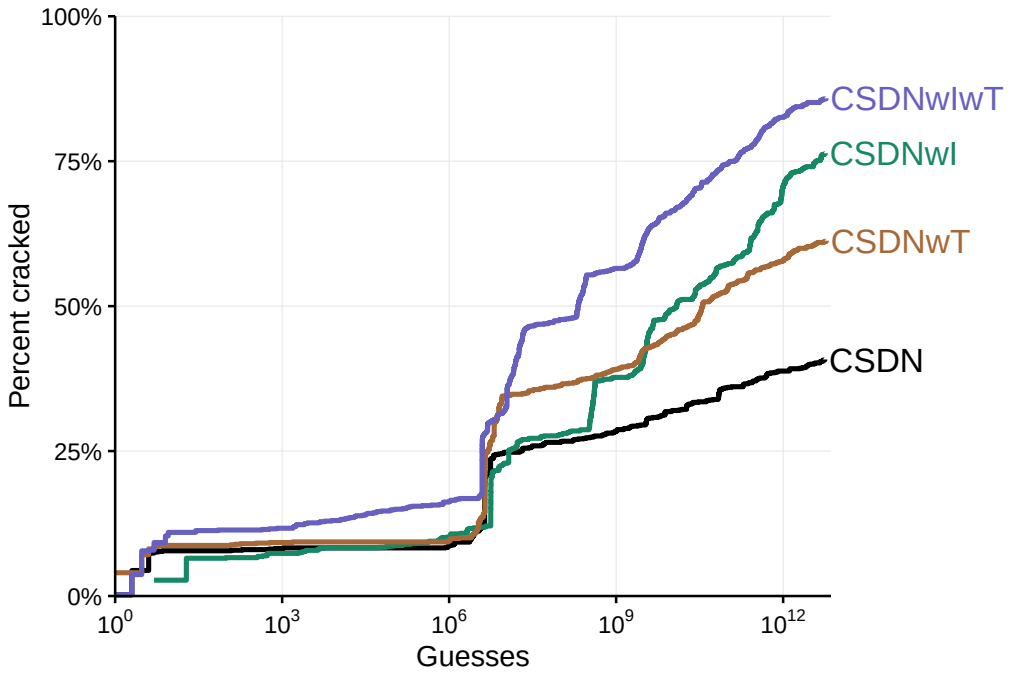


Figure 7.3: Guessing curves for CSDN passwords across all four experiments from Figure 7.2 (*wI* = with *Improved* model, *wT* = with additional training data)

thesis that were discussed in Chapter 5: quantized string frequencies, hybrid structures, and unseen terminal generation. A table of percent cracked for each condition is given in Table 7.1.

To get a better understanding of our results, we pull out the CSDN curves from Figure 7.2 into Figure 7.3. It is clear that both additional training data and framework improvements help with cracking CSDN passwords. Using pairwise

G^1 tests, all curves are significantly different except for the CSDNwI and CSDNwT curves ($p = 0.85$).

The wT models add all CSDN and Yahoo! passwords except for the 1,000 in the training set and this clearly outperforms the base condition after about 10^7 guesses. Interestingly, the CSDNwI model starts out slightly worse than the other models because it favors guessing RockYou passwords verbatim, but eventually overtakes the CSDNwT curve by guessing unseen terminals. The CSDNwIwT curve generally outperforms all other curves over the range of guess numbers.

Tuning for CSDN

CSDN is an interesting dataset. It has very different passwords, compared to English datasets like RockYou, since a large proportion of its passwords are all-digit strings [12, 84]. This leads to poor performance in early guessing across all four experiments, relative to RockYou and Yahoo!, as shown in Figure 7.2.

A noticeable aspect of our CSDN models is the sudden jump in cracked passwords around two million guesses. This is caused by training on RockYou and Yahoo!, which have many more six and seven-character passwords than CSDN, previously shown in Figure 7.1. Around two million guesses is where the guessing models start to guess eight-character passwords. This is a consequence of how we ran our experiment using a combination of different policies. CSDN has some six and seven-character passwords that get cracked, but our guessing model based on RockYou is not as efficient as it could be when attacking CSDN. This is realistic if we assume that our adversary has no knowledge of CSDN passwords, but even a modest sample would show that passwords less than eight characters are infrequent. An intelligent adversary might try to combine this information with their prior knowledge of passwords to mount a more powerful attack, perhaps by adjusting the guesses they make to better match the distribution of password lengths seen in their sample.

We can account for this using the guess-calculator framework. Figure 7.4 shows the curves of Figure 7.3 with one additional model: CSDN5000wI. This is an Improved model trained with RockYou and Yahoo! passwords and 5,000 additional CSDN passwords added only to the structures corpus.[2] This is much less training data than was given to the wIwT model, whose training data included over six million CSDN passwords, but with one crucial difference—we configured the framework to weight the 5,000 passwords so that their combined probability

[2] As described in Section 4.2, the structures corpus defines the length and character-class composition of a guessing model, while the terminals corpus describes the actual strings used to compose passwords.

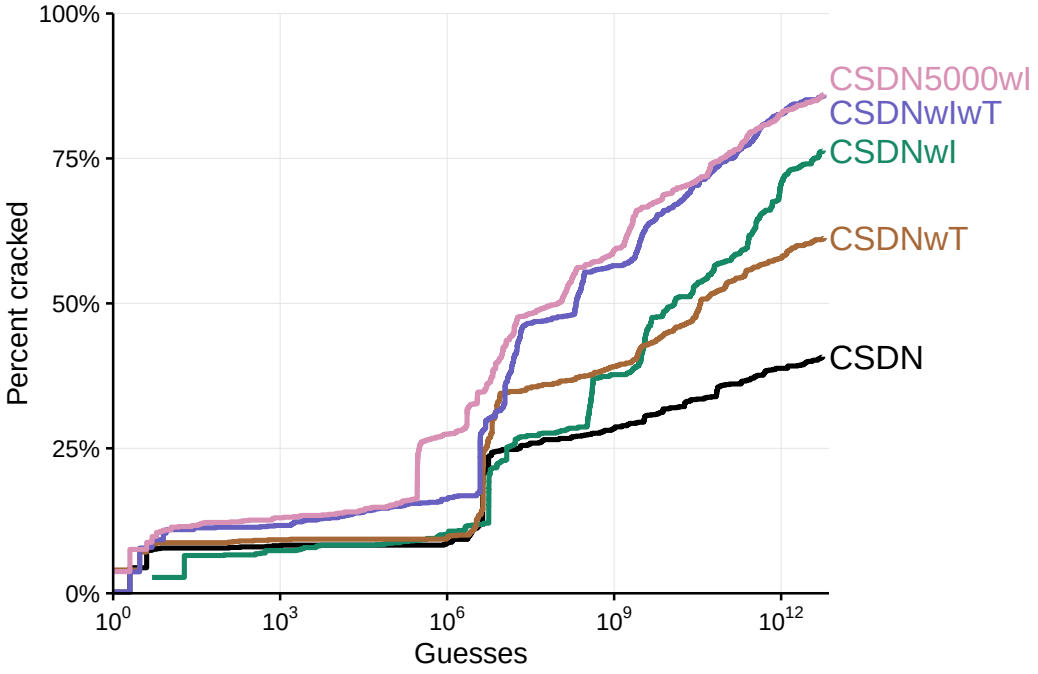


Figure 7.4: Guessing curves for CSDN passwords across all four experiments from Figure 7.2 (*wI* = with *Improved* model, *wT* = with additional training data) plus data from Experiment 8E (*5000wI* = with 5,000 additional CSDN passwords, see text for complete description).

mass was 10 times that of all other passwords in the training data.[3] This refines the model, tailoring it for CSDN passwords. Surprisingly, this small addition produces a model that is significantly better than all other models including the wIwT model (G^1 test, $p < 0.002$).

While the CSDN5000wI model might not be as efficient as a model built from training data that was manually curated, we believe this demonstrates that a more general, less manual approach to password strength evaluation can be useful.

Cracking Yahoo! Voices passwords

We also pull out the Yahoo! curves from Figure 7.2 into Figure 7.5. It is apparent that additional training data does not help to model Yahoo! passwords. This suggests that RockYou is a reasonable match for the Yahoo! dataset. In contrast, framework improvements help produce a significantly better model than the base or wT model ($p < 0.001$, G^1 test). Pairwise differences between all four curves were significantly different, with $p < 0.001$, except for the base and wT curves, and wT and wTwI curves, which were not significantly different ($p = 1.000$, G^1 test).

[3] In Section 6.2.2, we found that this performed best of the values we tried.

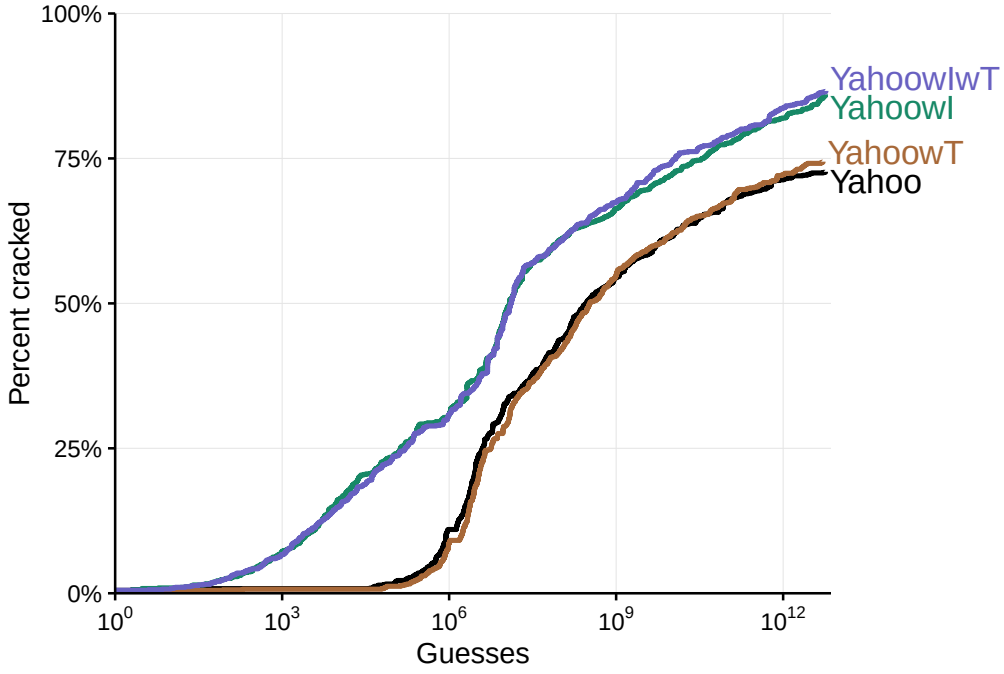


Figure 7.5: Guessing curves for Yahoo! passwords across all four experiments from Figure 7.2 (*wI* = with *Improved* model, *wT* = with additional training data)

	10^0	10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8	10^9
Yahoo	0.4%	0.7%	0.8%	0.8%	0.8%	1.6%	11.0%	32.7%	43.7%	54.4%
YahoowI	0.4%	0.9%	2.5%	7.1%	15.7%	23.2%	30.0%	47.1%	59.6%	64.7%

Table 7.2: Experiment 8A (top) & Experiment 8B (bottom) — Percent of sample cracked by each model for early guesses. For both models, guesses are made in probability order, but the model with improvements (YahoowI) cracks many more passwords early on because it has a better probability model.

The models with improvements are particularly strong in early guessing, below 10^6 guesses. Table 7.2 shows the y-axis values for Figure 7.5 for the Yahoo and YahoowI models. The Improved model clearly outperforms the Weir 2009 model —- after 32,175 guesses, the Weir 2009 model cracks 0.8% of passwords while the Improved model cracks 20.1%, over 25 times more. This gap narrows substantially as the number of guesses increases, since the Weir 2009 model eventually makes a large number of successful guesses. Note that if we had applied the Simpleguess algorithm from Chapter 3 we might have done even better at early guessing, but it is doubtful that we would have cracked more passwords overall.

Conclusion

In both the Yahoo! and CSDN datasets, we find that using improvements to the guess-calculator framework results in significantly better models over previous PCFG techniques. Given training data based on a single leaked password set, models built using framework improvements seem to outperform more naïve models built with substantially more training data. This suggests that we have found a more powerful learning algorithm, one that can make better use of data to produce more accurate models.

7.2 Evaluating standard-length passwords

In this section, we examine how the improvements we have made to the guess-calculator framework affect the evaluation of a complex, mixed-class policy that we call `4class8`, and compare this with a simple policy called `basic8`.[4] We show that the learning algorithm of Weir et al. [146] is not suited for mixed-class passwords and does a poor job of modeling early guesses correctly, i.e., high-probability passwords in the training data are not selected with high-probability by the resulting PCFG. In contrast, the improvements we have described to the guess-calculator framework produce a more accurate model of these passwords.

Test sets for both policies were taken from a random sample of the subset of RockYou that complied with the target policy. Both models were produced by training on our `Public` set with the test passwords held out. We provide configuration information for these experiments in Appendices B.21 and B.22.

7.2.1 Results

Figure 7.6 shows guessing curves for two PCFG models given the same training and test data: a model that mimics the *Weir 2009* learning algorithm, and an *Improved* model that incorporates the improvements from this thesis.

With the Improved model, using hybrid structures enables correctly learning the highest probability password in the training data. `P@ssw0rd` is the first guess made by this model and was the highest probability password in the training data. In contrast, the Weir 2009 model splits this password into 5 tokens, which lowers its probability enough that it does not try this password until guess number 350,584,956,955. The Weir 2009 model does not crack any passwords in the test set until guess number 1,499,611, because the tokenization penalty creates an inaccurate probability model. This issue is described below in more detail.

[4] See Section 2.4.4 for the naming convention we use to describe password-composition policies.

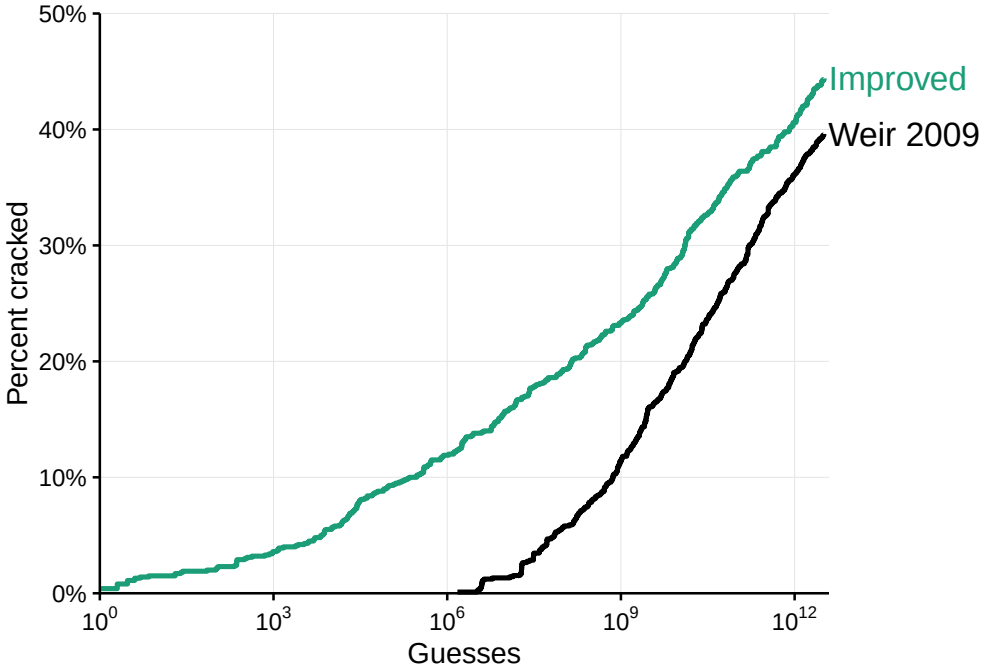


Figure 7.6: Experiment 9A — Evaluating two `4class8` models. The *Weir 2009* model used settings akin to Weir's original approach [146] when learning a PCFG, while the *Improved* model incorporates improvements from this thesis: quantized string frequencies, hybrid structures, and unseen terminal generation.

Recall from Section 5.4.2 that a tokenization penalty is applied based on the number of tokens in a password. For example, `Password` is parsed as a single token by a character-class tokenizer while `P@ssw0rd` is split into five tokens: `P`, `@`, `ssw`, `0`, `rd`. Table 7.3 shows the fifty highest-probability `4class8` passwords from the RockYou dataset. The minimum number of tokens in a `4class8` password is three, yet almost all of the passwords in Table 7.3 contain four or more tokens. This is what causes them to be inaccurately modeled by a standard character-class tokenizer. On the other hand, hybrid structures allow each password in the training data to be represented by a single token. The tradeoff is that these passwords are guessed twice, once by an untokenized structure and once by a tokenized structure. However, this experiment shows that the tradeoff is clearly worth it.

We can also examine the effect of our framework improvements on a `basic8` policy. This is shown in Figure 7.7. Again we see that the Improved model has much better performance than the Weir 2009 model. In this case, however, the improvements come mostly from the use of accurate string frequencies in learning the PCFG. Unseen terminal generation improves the model slightly as shown

1	P@ssw0rd	18	3edc#EDC	35	ZAQ!zaq1
2	!QAZ2wsx	19	#EDC4rfv	36	Hottie#1
3	1qaz!QAZ	20	12qw!@QW	37	Passw0rd!
4	1qaz@WSX	21	2wsx@WSX	38	@WSX3edc
5	!QAZ1qaz	22	Password1!	39	!@QW12qw
6	ZAQ!2wsx	23	Jesusis#1	40	7ujm&UJM
7	1qazZAQ!	24	P@55w0rd	41	P@ssw0rd1
8	!QAZxsw2	25	1941.Salembbb.41	42	0okm)OKM
9	Pa$$w0rd	26	@WSX2wsx	43	iydgTvm6d;yo
10	John3:16	27	!Qaz2wsx	44	@WSXxsw2
11	ZAQ!1qaz	28	Blink-182	45	1q2w!Q@W
12	zaq1!QAZ	29	ZAQ!xsw2	46	ZSE$5rdx
13	!QAZzaq1	30	#EDC3edc	47	BHU*8uhb
14	P@$$w0rd	31	zaq1ZAQ!	48	)OKM9ijn
15	1qazXSW@	32	Babygirl#1	49	0ki6;iiI
16	zaq1@WSX	33	!Q@W3e4r	50	Abc123!@#
17	z,iyd86I	34	Raiders#1		

Table 7.3: 50 most popular `4class8` passwords in the RockYou password set. Most passwords are composed of more than two character-class transitions.

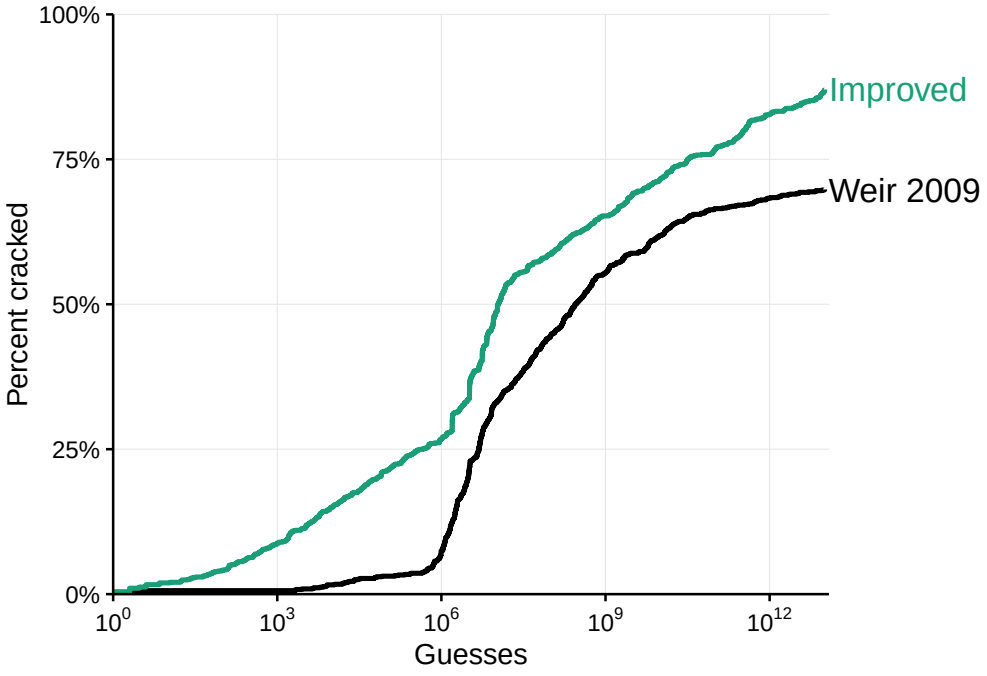


Figure 7.7: Experiment 9B — Evaluating two `basic8` models created in the same manner as Figure 7.6.

in Figure 5.3 and page 92, while hybrid structures seem to have little effect (see Figure 5.6(a) on page 103).

Conclusion

As described in Section 5.5, modeling high probability passwords in a `4class8` policy is quite difficult for a Weir PCFG, and the results here show that to be true. Note that, aside from unseen terminals generated by the Improved model, both models have the same language. That is, they both produce the same set of strings. Our framework improvements simply enable a better probability model for multiclass strings, as measured by the efficiency with which passwords from the test set are cracked. Figures 7.6 and 7.7 suggest that an approach similar to that used in this thesis is critical to evaluating the performance of both multiclass and simple policies in early guessing, where adversaries make a relatively small number of guesses.

7.3 Evaluating long passwords

In previous work, we found that password policies that require long passwords increase password strength more than complicated requirements, while also being more usable [67, 76]. In this section, we show how the improvements we have made to the guess-calculator framework affect our evaluation of these policies. Specifically, we take advantage of different tokenization schemes and the ability to load large lists of n-grams (multiword strings collected from the web) into the guess-calculator framework. We find that, over most of the guessing curve, a model built with a linguistic tokenizer provides similar or slightly worse performance compared to a model built on n-grams from the Google Web Corpus. However, after 10^{12} guesses the linguistic tokenizer model is superior. More appropriate sources of n-gram data might exist that perform better.

We examine the `basic16` policy, which requires a minimum of sixteen characters. This is a novel policy so we test only on passwords collected from Mechanical Turk. We collected these passwords using the methodology described in Section 2.2.

7.3.1 Results

Figure 7.8 shows how the improvements to the guess-calculator framework affect cracking with a `basic16` policy. Surprisingly, the use of many improvements that were helpful with other policies, such as unseen terminal generation and

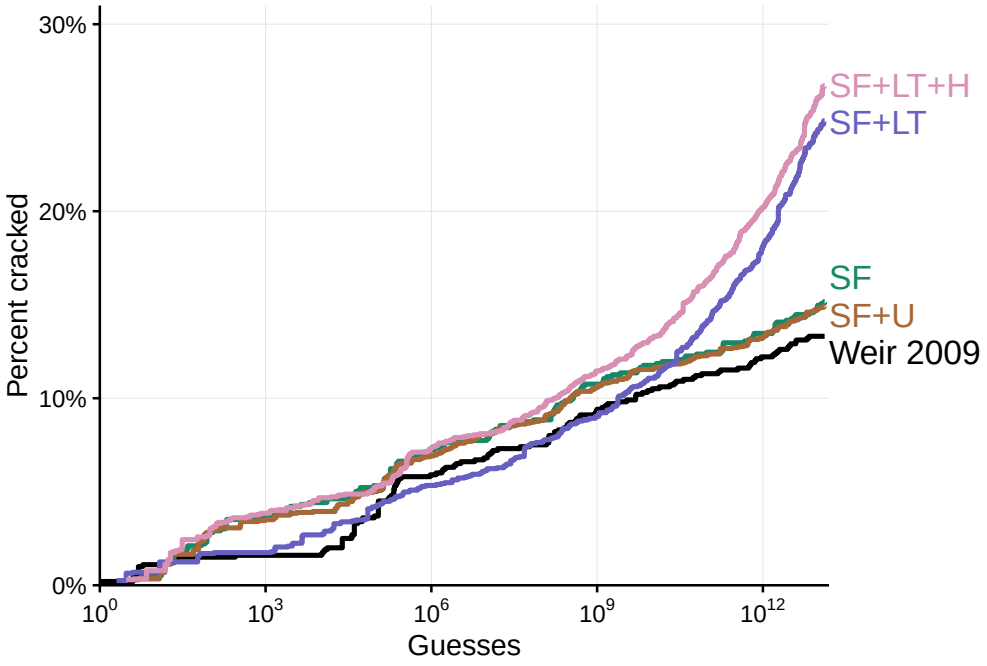


Figure 7.8: Experiment 4 — Evaluating the `basic16` policy using various configurations of the guess-calculator framework. The *Weir 2009* model used settings akin to Weir's original approach [146]. The other models enable various improvements: *SF* = quantized string frequencies, *U* = unseen terminal generation, *LT* = linguistic tokenization, and *H* = hybrid structures. The improvements are described in Chapter 5.

more accurate string frequencies, provide only modest improvements with long passwords.

Comparing the Weir 2009 and SF models, the SF model performs better but the difference is not striking. The SF model learns string frequencies, so this suggests that string frequencies used in long passwords that were cracked are more uniform than we observed with shorter passwords.

Enabling unseen terminal generation, shown in the SF+U model, seems to provide no improvement over learning string frequencies. This suggests that random terminals outside of the training data have a low probability of cracking long passwords. There are two factors that affect this. First, long passwords tend to have much longer terminals than short passwords. This is illustrated in Figure 7.9. This is important because the number of possible terminals increases exponentially with terminal length. Second, the amount of training data available for long terminals tapers off as terminals get longer, as shown in Figure 7.10. This illustrates that we do not have a corresponding exponential increase in training data for long terminals. Rather, we have a decrease in training data! The combination of these factors causes the space of unseen terminals to be exponentially larger

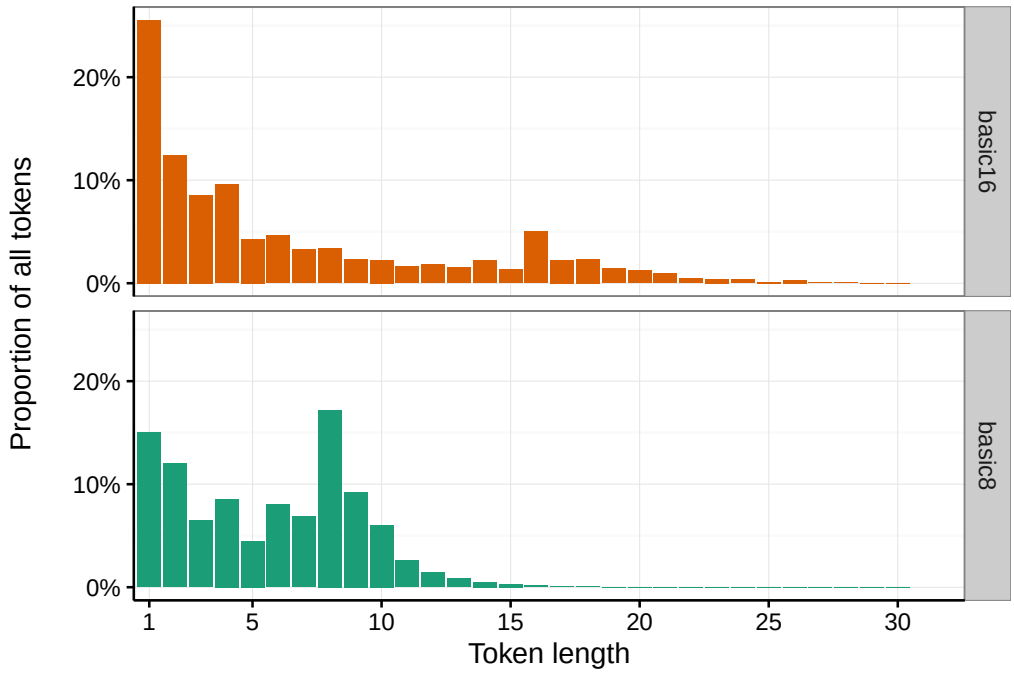


Figure 7.9: Comparison of token-length distributions for Weir-tokenized passwords from `basic8` and `basic16` policies. `basic16` passwords contain a larger proportion of very short and very long tokens than `basic8`.

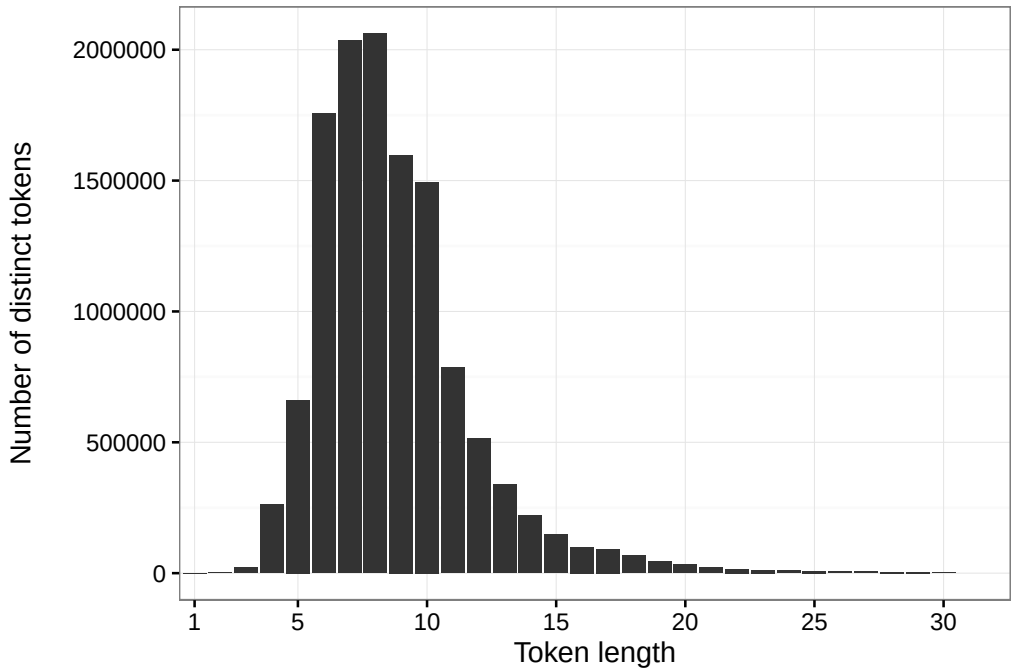


Figure 7.10: Number of distinct terminals available in `Public` training data per token length.

for long passwords compared to short passwords, with an expected reduction in cracking gains by brute-forcing unseen terminals.

Linguistic tokenization

The use of linguistic tokenization, shown in the SF+LT model, significantly improves cracking over other models (G^1, $p < 0.001$). Linguistic tokenization, described in Section 5.5.2, breaks long alphabetic strings into smaller tokens using a corpus of word n-grams. This enables the model to generate many more passwords than character-class tokenization. However, this also incurs a tokenization penalty, seen in the lagging curve of the SF+LT model in Figure 7.8. This curve cracks fewer passwords than other improved models until about 10^{10} guesses, when it overtakes those models. Using hybrid structures seems to mitigate the effect of the tokenization penalty, producing a model that is competitive with all other models in early guessing but also able to crack more passwords at large guess numbers.

We also evaluated a model with all improvements (SF+LT+H+Unseen). This model performed slightly better than the SF+LT+H curve, but not significantly so. For clarity, this curve is omitted from Figure 7.8.

7.3.2 Training with concatenated n-grams

Another approach to cracking long passwords is to try to overcome the decrease in training data depicted in Figure 7.10. To simulate how an adversary might do this, we return to the Google Web Corpus (GWC), a public set of n-grams scraped from the web [15]. The GWC contains phrases up to a length of five words that were seen 40 or more times, and each phrase is assigned a frequency. We take all alphabetic n-grams and concatenate them to create single strings like "allineedisyou."

We can use this corpus of strings as an input to the guess-calculator framework, replacing the unigrams in the `Public` training set described on page 137. In this experiment, we limited the strings to a length of 20 or less. We believe this provides enough training data for a proper experiment, without making the training set so large that it makes the experiment overly slow to run. The distribution of distinct tokens by length in this dataset is shown in Figure 7.11. The number of distinct strings in this corpus is 823,775,338.

The performance of this model is shown in Figure 7.12. Surprisingly, the model does not perform significantly better than other models we tested, suggesting that some sets of phrases scraped from the web might not be very good sources of training data for long passwords.

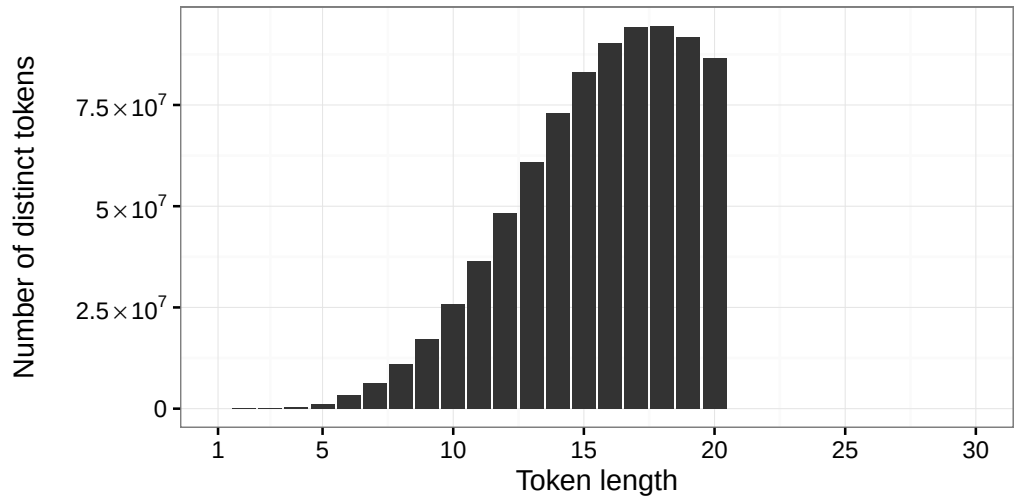


Figure 7.11: Number of distinct terminals available per token length after concatenating each n-gram in the Google Web Corpus. Only strings of length 20 or less were used (see text) but the x-axis is shown to length 30 for comparison with Figure 7.10.

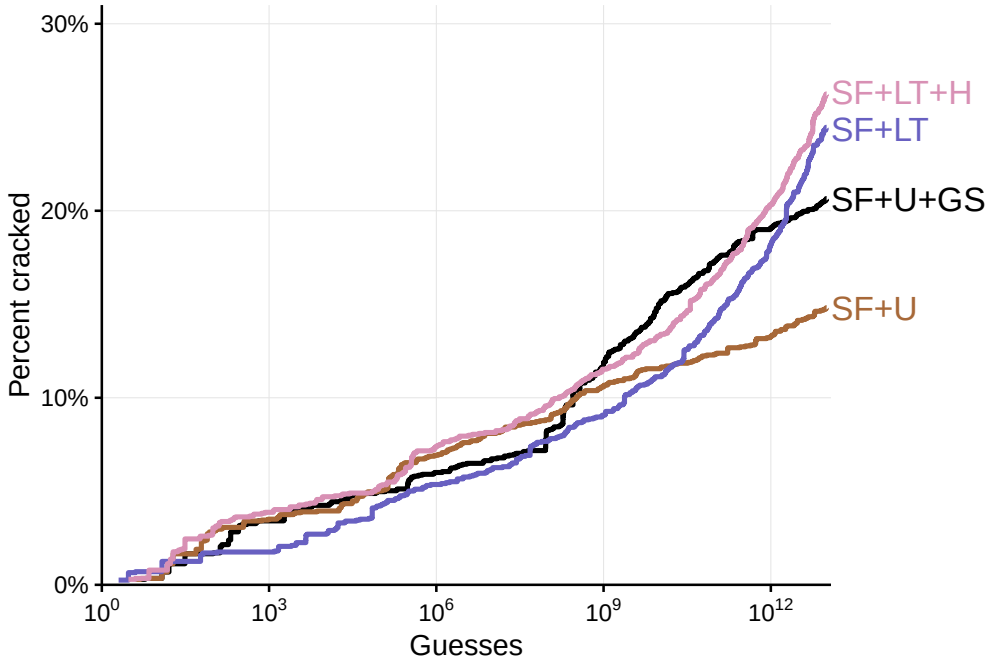


Figure 7.12: Experiment 4 & Experiment 10 — Three guessing curves from Figure 7.8 along with a guessing curve based on concatenated n-grams from the Google Web Corpus. We used all alphabetic n-grams whose concatenation was 20 letters or less in the *SF+U+GS* model.

1qaz2wsx3edc4rfv	fenderbabylove123	nintendosixtyfour
1q2w3e4r5t6y7u8i	martinmaaswilliam	JessikaRocks1234
zaq12wsxZAQ!@WSX	callielouprincess	ronniekarennicholas
passw0rdpassw0rd	virginiacaponera	vanessadebbie621
asdfghjklqwertyu	supernaturalpops	jenniferannnichols
passwordformoney	christinehalfmann	matthewisadoctor
passwordforturks	jillsarahchelsea	ElizabethDuthinh
lovelovekingking	stephanieringbauer	shelleyboockholdt
buttmunchbuttmunch	asdfghjklpoiuytr	passwordbreakeryes
stupidpasswordss	dhanalakshmikumar	dawnmichellespencer
featherwoodangel	cheryllynnharris	tootsieamberizzy
superlongpassword	jacquelineangelica	iloveandrewkiefer
november04052005	Samantha07161999	longislandmedium
december12121988	pangurbancricket	waltdisneymouses
tabathamartinez1	hellodollydarling	annaclairewagner
elizabethzagurski	cherryberrybanana	monkeyloveusmc20
hermionegrangerd	sixteenwordpassword	

Table 7.4: Experiment 4 — 50 passwords cracked by the SF+LT+H model and not by the SF+U+GS model after making 10 trillion guesses with each model. Passwords are arranged in columns, with the earliest passwords in the leftmost column, in the order that they were cracked. In contrast to Table 7.5, most passwords are names or nonsense phrases.

To help understand the differences between a model using linguistic tokenization and one based on a large corpus of n-gram training data, we consider two particular models: SF+LT+H and SF+U+GS. We examine the first 50 passwords cracked by each model that were not cracked by the other model. These passwords are presented in Tables 7.4 and 7.5. We find that the SF+LT+H model is able to crack passwords built with nonsense phrases, like "feather wood izzy," or names that do not exist in the GWC, like "cheryl lynn harris." In contrast, the SF+U+GS model is able to crack passwords that include recognizable phrases, like "hail to the victors," or place names, like "hoboken new jersey."

Conclusion

Overall, we find that adding concatenated n-grams to the training data does not significantly improve cracking performance over a more naïve approach involving linguistic tokenization. This was the case even though we added a huge number of n-grams, over 823 million, where each n-gram had been seen 40 or more times on the web [15]. It is possible that web n-grams are a particularly poor source of training data for passwords. We did not experiment with other sources of n-grams.

Qwertyuiopasdfg!	onethingaboutmusic	mypetsaregreat24
utahsaltlakecity	myfavoriteanimeare	irunlikethewind!
hailtothevictors	iliveinphiladelphia	alvinstonflyers11
hobokennewjersey	helpmeouthereplease	Fearisthemindkiller
cannedgreenbeans	thisisfuckingstupid	Iamwritingaproposal
thatswhatshesaid	youmakeallthingsnew	boysetsfireboysets1
thatiswaytoolong	stuyvesanthighschool	accenturemumbai10
neckneckneckneck	freeonlinefreeonline	ramyasrinivasan01
lutefiskandlefse	fairbanksmorseengine	thetimehascome666
allofthemaremine	pawcatuckconnecticut	Swagtastic123456
becauseweseparate	shamikghosh12345	harristeeter1111
thebrowncowjumped	threebeansalad12	itstherealthing87
tampabaybuccaneers	BINGHAMTONBEARCATS	spiderman5908577
everymanforhimself	jovanjovanovic123	gruetlilaager2010
surveysurveysurvey	shantanusharma123	christopherbeckham3
bitemebitemebiteme	hotchilipeppers123	hamburgerhotdog32
thisistotallycrazy	actuaryactuary17	

Table 7.5: Experiment 10 — 50 passwords cracked by the SF+U+GS model and not by the SF+LT+H model after making 10 trillion guesses with each model. Passwords are arranged in columns, with the earliest passwords in the leftmost column, in the order that they were cracked. In contrast to Table 7.4, many passwords are places or recognizable phrases.

Nevertheless, the n-gram model was superior for some parts of the guessing curve. A more complex hybrid approach might work best on long passwords, in which concatenated n-grams are employed for early guesses but structures learned from linguistic tokenization are also available to the model to construct more exotic passwords. We have not evaluated such an approach.

7.4 Applying survival analysis to passwords

The data and results for this section were drawn from the paper "Measuring password guessability for an entire university" authored by several members of our research group [95]. The data was collected on machines to which we did not have direct access, and was subsequently destroyed. Therefore, we are unable to reevaluate this data using the improvements to the guess-calculator framework.

My contributions to this paper included writing, work on what would eventually become the guess-calculator framework developed in this thesis,

> *and scripts for performing automated statistical analysis. The following material will not appear in the thesis of any other student.*

In collaboration with our Information Technology division, we were provided with the opportunity to evaluate a complex password policy currently in use at Carnegie Mellon University (CMU). Though we did not have direct access to the passwords themselves, we could perform statistical analyses on them, compare them to other password sets of our choosing, evaluate them with the guess-calculator framework and publish the results of our findings. The experiments we selected compared passwords used at CMU to password subsets from leaked datasets and passwords collected from online studies. An important feature of our analysis is the use of survival analysis to compare guessing curves, and to discover demographic and behavioral factors that are correlated with password strength.

First we describe our training and test datasets, then we discuss two analyses: evaluating password sets under a complex policy, and identifying factors that are correlated with password strength.

7.4.1 Target policies

All of the policies we evaluated share a similar password-composition policy. It can be described in the following way:

> This policy requires a minimum of eight characters including an uppercase and lowercase letter, a symbol, and a digit. The letters in the password, when concatenated together, must not equal a dictionary word.

When collecting password samples from Mechanical Turk, we described the policy using this language.

We refer to the set of passwords that we evaluated at Carnegie Mellon as CMUactive. We also had indirect access to inactive passwords from the Carnegie Mellon database, and we refer to this set as CMUinactive. We used the latter set only for training. We compared the CMUactive set to the following sets of passwords:

MTsim
: 1,000 passwords collected from an MTurk experiment designed to simulate CMU password creation as closely as possible, both in policy requirements and website design. This matches the `andrew8` policy we described in Section 2.4.4.

MTcomp8
: 1,000 passwords collected from MTurk, matching the `comp8` policy we described in Section 2.4.4.

RYcomp8
1,000 plaintext passwords from the RockYou dataset described in Appendix B.4.3. These passwords were sampled from 42,496 `andrew8`-compliant passwords.

Ycomp8
1,000 plaintext passwords from the Yahoo! Voices dataset described in Appendix B.4.3, sampled from 2,693 `andrew8`-compliant passwords.

CSDNcomp8
1,000 plaintext passwords from the Chinese Developer Network dataset described in Appendix B.4.3, sampled from 12,455 `andrew8`-compliant passwords.

SFcomp8
1,000 cracked passwords from Strategic Forecasting, Inc., also known as Stratfor, described in Appendix B.4.3, sampled from 8,357 `andrew8`-compliant passwords.

Gcomp8
896 cracked passwords from the Gawker dataset described in Appendix B.4.3. There were only 896 `andrew8`-compliant passwords.

Only one dataset, MTcomp8, differed from the others in using a larger dictionary for its dictionary check.

7.4.2 Evaluating potential proxies

We first examine the password datasets described above to see how closely they match the CMUactive set. This is an important investigation. Password-policy evaluation requires samples from the target to evaluate, but it is unclear where to get these samples. Most organizations do not have access to cleartext passwords as we did in this experiment. Therefore, we used our access to these passwords to see how well other password sets compare. This provides evidence in favor of potential proxies.

Our evaluation has two parts. First, we examine guessing curves for our datasets to see how closely they match. Second, we compare statistics on password composition, such as length and mean numbers of digits and symbols.

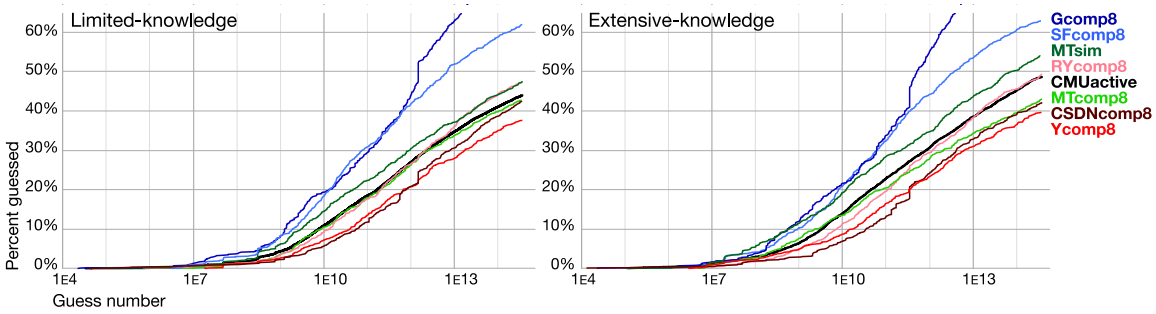


Figure 7.13: Experiment CMU-1 & Experiment CMU-2 — Guessing curves for each of the policies described in Section 7.4.1. The graph on the left represents an adversary with only public data. The graph on the right represents an adversary with many passwords from the target policy.

Comparing guessing curves

We examine guessing curves under two guessing models.[5] The limited-knowledge guessing model was trained using the `Public` dataset, plus the non-free Openwall dataset [109], and the remaining passwords from the CSDN, Stratfor, and Gawker datasets after holding out passwords for testing. Details of the configuration of this training set are provided in Appendix B.26. The extensive-knowledge model started with the training data for the limited-knowledge model and added 20,000 CMUactive passwords and 15,000 inactive CMU passwords. This configuration is detailed in Appendix B.27.

This analysis was performed before we developed the improvements in Chapter 5. However, the analysis methodology could easily be applied to guess numbers collected using newer guessing models. We use the G^1 test from Section 6.1.4 to compare the guessing curves of these datasets.

Figure 7.13 shows the results of our evaluation with the limited-knowledge and extensive-knowledge models. Holm-Bonferroni corrected p-values from the G^1 test find that the Yahoo!, Stratfor, and Gawker sets are all significantly different for the limited-knowledge model, and the MTsim and CSDNcomp8 sets are also significantly different under the extensive-knowledge model. These results suggest that the RockYou set and MTcomp8 can be used as a reasonable proxy for passwords from the target policy. The evaluation also confirms the negative utility of datasets of cracked passwords for policy evaluation—both Gcomp8 and SFcomp8 were much easier to guess than the target policy. While this might be an obvious finding, there are many sets of passwords that are released to the Internet

[5] We actually evaluated several different models that are discussed in our paper [95], but for the purpose of this section we consider two models on either end of the spectrum of training data.

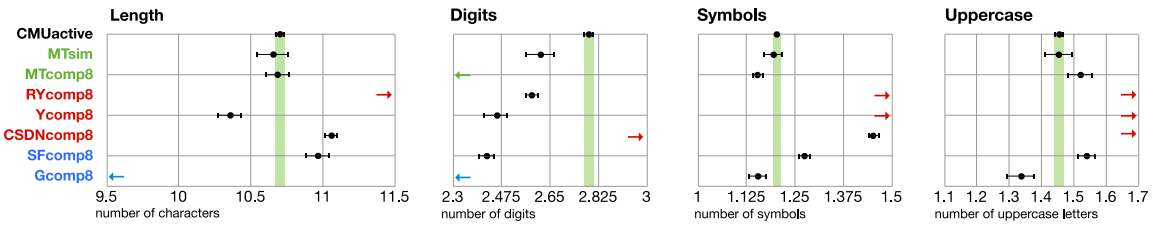


Figure 7.14: Aggregate password-composition statistics, by password set, with 95% confidence intervals. The confidence interval for CMUactive is shaded green. MTsim is generally the closest match for CMUactive.

	Observed probability of nth most popular password									
	1	2	3	4	5	6	7	8	9	10
CMUactive	0.094%	0.051%	0.043%	0.039%	0.035%	0.035%	0.031%	0.027%	0.024%	0.024%
MTsim	0.200%†	0.100%†	0.100%†	0.100%†	0.100%†	0.067%†	0.067%†	0.067%†	0.067%†	0.067%†
MTcomp8	0.233%†	0.100%†	0.067%†	0.067%†	0.067%†	0.067%†	0.067%†	0.067%†	0.067%†	0.067%†
RYcomp8	0.513%	0.304%	0.242%	0.214%	0.134%	0.115%	0.101%†	0.099%†	0.068%†	0.066%†
Ycomp8	0.520%	0.149%†	0.149%†	0.149%†	0.111%†	0.111%†	0.111%†	0.111%†	0.111%†	0.111%†
CSDNcomp8	2.529%	1.429%	0.715%	0.426%	0.241%	0.233%	0.225%	0.217%	0.161%	0.128%
SFcomp8	0.191%†	0.191%	0.191%	0.191%	0.179%	0.168%	0.096%†	0.096%†	0.084%†	0.084%†
Gcomp8	4.911%	0.893%	0.893%	0.893%	0.670%	0.670%	0.670%	0.558%	0.558%	0.558%

Table 7.6: Empirical probabilities for the 10 most popular passwords. Probabilities that are not significantly different from CMUactive for a given password rank are grayed out and marked with a † (Holm-Bonferroni-corrected χ^2 test, $p < 0.05$). All other policies are weaker than CMUactive, but MTsim and MTcomp8 are the closest on this metric.

without provenance. In other words, it might not be clear whether a particular dataset consists of cracked passwords or passwords that were stored in plaintext. The Gcomp8 and SFcomp8 curves underscore the importance of having a set of reliable passwords for policy evaluation.

Comparing aggregate statistics

We also examine several aggregate password statistics across the datasets. Figure 7.14 shows how the average length and number of digits, symbols, and uppercase characters per password for each dataset compares to that of the CMUactive policy. Confidence intervals in the figure were produced using the bootstrapping technique described in Section 6.1.3. Examining these properties, we find that MTsim is consistently close to CMUactive. Surprisingly, RYcomp8 is quite far from CMUactive, despite having a very close guessing curve.

Finally, Table 7.6 presents the probabilities of the top 10 most frequent passwords in each dataset. This metric indicates the strength of the 10 weakest passwords in each dataset, and is a small measure of the password-probability

distribution.[6] We compared the probabilities for each dataset to the CMUactive set using a Holm-Bonferroni corrected χ^2 test. This is not a rigorous comparison; it merely provides us with additional data on which datasets are similar to CMUactive. We find that all other datasets are weaker than CMUactive, but the MTsim and MTcomp8 datasets are consistently close.

Conclusion

Our analysis finds that the RYcomp8, MTsim, and MTcomp8 datasets seem to be reasonable choices to use as proxies, though none are a perfect match for the target. In the future, we hope that other researchers will repeat this analysis with other policies to expand our knowledge of appropriate password sources.

7.4.3 Factors correlated with password strength

We were fortunate to have demographic information available for the users of CMUactive passwords. Using the Cox regression technique discussed in Section 6.1.4, we can identify factors that seem to be correlated with password strength.

The same care should be taken with Cox regression as with other regression analysis methods. In the absence of a randomized, controlled experiment, we cannot make causal conclusions. Further, since many of the independent variables in our data set are correlated with other independent variables, there might be factors correlated with password strength that did not get identified by the regression.

This last point is especially true since we employed a model selection technique. Model selection, also called feature selection, reduces the number of factors used in a model when some factors do not improve the model very much. This is intended to reduce overfitting and present a model with greater external validity. There were a large number of demographic factors available to us in this study, so we used model selection to decrease the number of factors selected by the regression. We use a method known as stepwise backward elimination with BIC (Bayesian Information Criteria) to select between models [119]. This approach has been superseded by more advanced approaches, however, so we no longer recommend it. Instead, we recommend a regularized regression technique in which cross validation is used in producing the final model. Packages that implement this method are readily available in R [50,130].

[6] Recall from Section 1.2 that we cannot get a good picture of the entire distribution. We only have confidence in the most frequent passwords.

Factor	Coef.	Exp(coef)	SE	p-value
login count	<0.001	1.000	<0.001	<0.001
password fail rate	-0.543	0.581	0.116	<0.001
gender (male)	0.078	0.925	0.027	0.005
(college) engineering	-0.273	0.761	0.048	<0.001
(college) humanities	-0.107	0.898	0.054	0.048
(college) public policy	0.079	1.082	0.058	0.176
(college) science	-0.325	0.722	0.062	<0.001
(college) other	-0.103	0.902	0.053	0.051
(college) computer science	-0.459	0.632	0.055	<0.001
(college) business	0.185	1.203	0.054	<0.001

Table 7.7: Experiment CMU-3 — Cox regression results for personnel with consistent passwords. Negative coefficients indicate stronger passwords. The exponential of the coefficient (Exp(coef)) indicates how strongly that factor being true affects the probability of cracking compared to the baseline category, which is implicitly coded as false. The baseline category for gender is female and for college is fine arts. For example, the fourth line indicates that, with other factors held constant, engineering personnel are 76.1% as likely to have their passwords guessed as fine arts personnel.

Results

We performed our analysis on the subset of active CMU users who kept the same password for several months. We call these *consistent* personnel. We make this restriction so that password-related factors such as fail rate and login count can be tied to the specific password whose guess number we assigned. We explore other subsets of CMU personnel in our paper [95].

Table 7.7 shows the result of our Cox regression analysis on these users. We find that various colleges and gender are strongly correlated with password strength. Our results for gender, that men have slightly stronger passwords than women, was also found by Bonneau in his analysis of Yahoo! passwords [7]. We also find that users who make more password errors (password fail rate) have stronger passwords than other users, and users who log in more often have slightly weaker passwords. An additional password error per login attempt is associated with a password only 58% as likely to be guessed. Each additional login during the measurement period is associated with an estimated increase in the likelihood of guessing of 0.026%. Though this effect is statistically significant, we consider the size of this effect to be negligible.

Conclusion

The regression results highlight some interesting correlations between password strength and demographic and behavioral factors that had not been measured previously. Though we should be careful not to draw causal conclusions, we hope that other researchers will use survival analysis methods to uncover more insights into how various factors are related to the strength of user-selected passwords.

Chapter 8

Conclusion

In this chapter, we provide a summary of our main contributions in Section 8.1, discuss the limitations of our work in Section 8.2, and present some ideas for future work in Section 8.3. Since our work might enable malicious parties to crack passwords more quickly, we conclude this thesis with a discussion of its potential societal impact in Section 8.4.

8.1 Contributions

The contributions of this thesis are described below.

The guess-calculator framework

We developed a system that automatically learns a guessing model for passwords based on a training set, and uses this model to assign guess numbers to passwords in any number of test sets. The model is learned using Weir's probabilistic context-free grammar (PCFG) [146]. Our implementation takes advantage of unique features of the grammar to assign guess numbers in a manner that operates much faster than explicitly enumerating individual guesses. We provide detailed algorithmic descriptions of the novel parts of the framework in Chapter 4 and also make the source code publicly available.[1]

Improvements to the guessing model

In Chapter 5, we make several improvements to Weir's PCFG to model a more sophisticated adversary. These improvements include more accurate probabilities for strings, generating unseen terminals, linguistic tokenization, and hybrid

[1] `https://github.com/cupslab/guess-calculator-framework/tree/v1.0.0`

structures. In addition, we improve the implementation of the guess-calculator framework so that it can handle large grammars, including large lists of word n-grams. We find that these improvements greatly increase the efficiency of our guessing models, especially with complex policies. In contrast to previous work, we show that a PCFG-based model using hybrid structures can crack a significant proportion of passwords even at low guess numbers. This makes this methodology potentially relevant to online as well as offline-attack threat models.

In Chapter 3, we examine other approaches to password cracking. We find that, for a particular policy, our automated approach compares favorably to the results of professional password crackers.

Methodological guidance

In Section 6.2, we show how the above improvements to the framework affect the evaluation of policies, which provides insights into what training data should be used and how the framework should be configured for various policies. We find that leaked password sets seem to be good sources of training data, and synthetic datasets like those produced by Openwall [109] are poor. We also find that including samples from the target policy in our training data improves the efficiency of the resulting models, and that different weightings do not affect the result very much.

In Section 6.1, we provide guidance on the analysis of data collected using the guess-calculator framework. This guidance includes an examination of existing statistical methods, including methods from survival analysis, and their applicability to guessing data.

Case studies

Finally, we use the guess-calculator framework to evaluate various policies in Chapter 7. We find that we can produce significantly better guessing models than previous PCFG methods, for a wide range of policies. In Section 7.1, we present one of our key results—guessing models can be easily tuned to a specific target using a few thousand samples. Finally, in Section 7.4, we show how survival analysis can be used together with guessing data, demographic, and behavioral data to compare datasets and examine correlations between password strength and other factors.

8.2 Limitations

Guess-calculator framework

Our framework requires a plaintext sample of passwords to use as a test set, though authentication systems should never store passwords in this way. This introduces a barrier to use of the framework by researchers and system administrators who might only have hashed passwords to evaluate. The framework was not designed for this use case. Removing this limitation would require that guesses be made explicitly against a password file in order to evaluate a policy, and this would require significantly more time to run than using the framework.

Guessing models

Our methodology assumes a very specific threat model, described in Section 1.1. Other threat models are not modeled in this thesis, such as online attacks. The Simpleguess algorithm, presented in Section 3.1, is a better choice than our technique for modeling online-attack scenarios, and even better methods might exist.

Our threat model also has some limitations that might not be evident at first glance. We assume that the adversary does not have information about specific passwords that could be used to tailor guesses against specific users. We are also unable to model password policies accurately if they cannot be captured by our restricted PCFG. Such policies might forbid specific passwords or particular substrings within passwords. These policies can still be evaluated by generating strings directly and removing guesses which are forbidden by the policy, but this rules out the lookup improvements described in Chapter 4.

Password policies might also be modeled better by specialized approaches outside the scope of this thesis. This thesis models passwords using a restricted probabilistic context-free grammar (PCFG). Markov models [88], more sophisticated grammars [115], or deep learning approaches [100] might produce more accurate models than our approach. However, even these learning algorithms might be limited in their ability to model the true knowledge of adversaries.

Similarly, professional crackers use cracking tools that primarily employ "mangling" to create new passwords [54]. Mangling is a generic term used to describe various transformations that can be applied to an input password set, such as appending random numbers or substituting letters [93]. While many of these techniques could be modeled with a probabilistic context-free grammar, they would not be captured by the limited grammar used in this thesis. Therefore, it

might be possible for an adversary to crack passwords that would not be cracked efficiently by the guess-calculator framework.

Other attacks

Passwords are vulnerable to many other types of attacks besides guessing. Many organizations store passwords in plaintext [9], so when these password files are stolen the adversary does not need to guess passwords. Passwords can also be revealed through phishing attacks [41]. These attacks completely ignore password-composition policies and require other security processes that are not covered by this thesis.

Relevance

We assume that password disclosure is an important problem and will continue to be so for the foreseeable future. However, we do not attempt a rigorous examination of this question. Password strength might be irrelevant in the near future, e.g., through the widespread adoption of two-factor authentication systems. Password disclosure itself might have a low risk or low value to others so that interventions are unnecessary. Finally, even though a large number of data breaches have been reported, it is possible that adversaries do not spend much effort on guessing attacks against password data. Instead, they might be more interested in other details of users that can be recovered during a breach, such as email addresses, real names, or social security numbers.

8.3 Future work

There are many ideas related to this thesis that we did not have time to address or are outside the scope of this work. We list many of these ideas below, as pointers to those interested in expanding this work in the future.

8.3.1 Inputs to the framework

Our framework evaluates password policies using a training set that represents adversary knowledge about a target policy, and a test set of passwords from the target policy. For both sets, we explored only a small number of the possible choices. The methodology used in this thesis could be used to conduct experiments aimed at finding training sources that crack more passwords for given test sets. For common policies like `basic8`, we found in Section 7.2 that leaked datasets like the RockYou and Yahoo! Voices datasets work well. In Section 7.3, however, we

found that rare policies like `basic16` are difficult to collect training data for. We explored using concatenated n-grams from the Google Web Corpus as an input, but we have only scratched the surface with respect to sources of n-gram data and there are many more sources that one could try.

When selecting a test set, the best option is to randomly sample passwords from the target policy. In previous work, we found that passwords collected from Mechanical Turk might be reasonable proxies for samples from the target policy [95], but this work should be replicated with the improved guess-calculator framework and other proxies should be investigated. To evaluate test sets, it would be necessary to collect some samples from the target policy and compare these samples to proxies from other sources. We suggest the methodology described in Section 7.4 and previously published in Mazurek et al. [95] to perform this comparison. It uses survival analysis to measure the similarity of guessing curves (using the methods described in Section 6.1.4), and also compares other properties of passwords as an additional check.

The framework also has a number of parameters that could be adjusted. Future work could examine how changing weights of samples and public data affect the resulting models, for a more diverse set of policies than we tested in Section 6.2.2. One could also examine the addition of an explicit penalty on tokenized structures. The findings of Section 5.5.3.2 indicate that cracking passwords with untokenized structures is quite preferable to using tokenized structures, so penalizing tokenized structures so that they are tried later might improve our guessing models.

In all of these cases, researchers should be cautious not to overtrain guessing models to a particular test set. Standard practices from machine learning are advisable, such as the use of multiple holdout sets.

8.3.2 Higher guess cutoffs

The current implementation of the framework stores patterns explicitly in a lookup table. For large guess cutoffs, the size of this table can become restrictive. For certain sets of inputs, a lookup table for 10^{13} guesses can require 2 terabytes of space, so creating a lookup table for 10^{15} guesses would be prohibitively expensive for a typical researcher or IT department, even though we know that professional password crackers can easily reach this number of guesses.[2]

One could address this problem by altering the lookup table implementation to trade accuracy for space. Rather than store patterns explicitly, one could alter the table generation process to store the number of guesses in each of several

[2]While we expect that storage will become cheaper over time, the desired guess cutoff is also likely to increase as adversaries take advantage of cheaper computing power.

probability bins. For example, one could bin probabilities logarithmically so that bins cover the intervals $[1, 0.5]$, $(0.5, 0.25]$, $(0.25, 0.125]$, and so on. The framework would still iterate over all patterns using intelligent skipping but instead of storing a separate record for each pattern, the framework would add the number of guesses associated with that pattern to its respective bin. This would greatly reduce the amount of space needed for the framework to run, allowing much higher cutoffs to be reached. On lookup, the framework would assign every password in a bin the lowest guess number for that bin, leading to a slight underestimate of security: passwords would be considered cracked before they would actually be cracked by a model. However, one could shrink the bin sizes as desired to increase accuracy, or use interpolation to select a guess number within the range of a bin.

This approach has a few drawbacks. First, it will be less accurate with regard to the guess numbers assigned to individual passwords. Second, it is not possible to generate guesses explicitly from the lookup table. We can use the current lookup table implementation to generate guesses explicitly, by enumerating the guesses in each pattern, but this is no longer possible if patterns are not stored. We also use patterns to detect errors in step A4 of Algorithm 4.7A, and we would lose this ability if patterns were not stored. Nevertheless, altering the lookup table implementation could be used to reach much higher guess cutoffs than we have explored previously, and its accuracy could be checked against the current method at shorter cutoffs.

8.3.3 Modifying the learning phase

Once input datasets are weighted and combined, the framework learns a guessing model from the data. In the current framework, we learn a Weir PCFG from this data, along with various improvements described in Chapter 5. There are many ways that we could modify the learning step of the framework and potentially improve our models.

Modified structures

Sticking with a Weir PCFG, one could look at new ways of parsing the input data. For example, one could "mangle" the input passwords [93], e.g., substitute numbers for letters, to allow the guessing model to crack a more diverse set of passwords than it is currently capable of. While the unseen-terminal-generation improvement allows our framework to crack passwords with unseen terminals, it is still unable to crack passwords with unseen structures. Mangling structures is

one way to address this issue, and matches a method used in popular cracking tools.

In Section 7.3.2, we found that using concatenated n-grams in training data helps in cracking certain passwords more quickly than the linguistic tokenization approach, but is unable to crack passwords that contained combinations of words not seen in the training data. An extension to this approach is to learn three-way hybrid structures: untokenized structures, Weir-tokenized structures that would get filled with concatenated n-grams, and structures learned using linguistic tokenization. Such a model might be superior to the current models we produce for long passwords.

Finally, we were unable to get good results from unsupervised tokenization. We enumerate a few possible reasons for this at the end of Section 5.5.3.2, and future work might include a return to this approach to find potential improvements.

Other linguistic models

Future work could also look at alternatives to a Weir PCFG. The Weir PCFG is a restricted grammar, as described in Section 2.3.1.1, and one could consider more complex PCFGs. The approaches referenced in Section 2.7 might produce better models than current approaches. Unfortunately, the implementation used in this thesis is highly optimized for a Weir PCFG, so it cannot be easily extended to more complex grammars.

Recurrent neural networks (RNNs) have recently been used with success in linguistic modeling [101,102,132], and they seem applicable to password modeling as well. Markov-model approaches are similar, but RNNs are more powerful because they can encode an arbitrarily large "history" [132]. This might be especially useful with long passwords.

Finally, we are not aware of any use of ensemble methods in password modeling. An ensemble method combines multiple probability models to produce a single model that is often more accurate than any of the models alone [42]. Multiple PCFGs, or PCFGs and other linguistic models, could be combined into an ensemble whose performance might be better than current models.

The ideas above represent our intuition about how the guess calculator could be improved or extended. Another important idea for future work is to collect data about password guessing attacks in the wild. Many of our ideas about guessing attacks come from anecdotes or professional password crackers, whose methods might not match those of true adversaries. Collecting data about attacks in the

wild would help us move our research efforts closer to reality, and might uncover holes in our methodology that have been missed.

8.4 Societal impact

If we assume that adversaries' tools are not as good as what this thesis has produced, adversaries might use this code to develop more efficient guessing attacks. These attacks will still fail against users who use best practices when it comes to passwords: different, randomly generated passwords of reasonable length for every site. Unfortunately, we know that many users do not do this, and researchers have made the case that users cannot do this without a password manager [49]. Thus, this thesis could make users who do not follow these practices more vulnerable. The impact of this thesis might then be largely negative: more users getting hacked, or a huge cost in time as many users migrate to password managers and replace their reused, easy passwords with randomly generated ones. This is not necessarily the extent of the negative impact, as these password managers might have unforeseen vulnerabilities that put people at more risk than current password authentication systems.

However, exposing vulnerabilities can have an ultimately positive effect. Remember that this thesis assumes that an adversary is making a guessing attack, and these attacks are far more effective against offline targets than against online ones. We hope that quantifying the vulnerability of passwords to guessing will encourage organizations to rethink their password policies and improve their authentication systems' security. For example, should follow best practices with regard to the storage of passwords. Some of these practices are summarized in [48] and impose no cost on the user. Similarly, supporting two-factor authentication can also reduce risk. This comes with a usability cost, but the security benefit can be worth the tradeoff.

Since the guess-calculator framework can be used to quantify the strength of many common password policies that rely on character-class requirements, system administrators could take this information into account when making decisions about the security and usability of their authentication systems. We found that over 10% of six-character-minimum CSDN passwords can be cracked in just 10 guesses. We hope that system administrators can incorporate information like this into a risk analysis framework to decide which security mitigations to put into place. Administrators could also use the guess-calculator framework itself to analyze security at their organization. Even though there are still many unanswered questions, particularly with regard to more complex password policies, we hope

that this thesis will expand the field of authentication research and help answer these questions more definitively in the future.

Appendix A

Additional Functions

In this section, we present a few functions that we need to call in the course of computing the guess number of a password. While we have tried to provide details on all of the algorithms we developed, we played no part in the development of these formulas so we do not devote much space to explaining them.

A.1 Permutations of a multiset

The standard formula for permutations of a multiset is given by:[1]

$$\frac{\mathtt{n}!}{\mathtt{m}_1!\mathtt{m}_2!\ldots\mathtt{m}_k!} \tag{A.1.1}$$

We define the function `PERMS-TOTAL(a)` which takes an array with a given permutation, e.g., `a = [0, 0, 2, 3]`, and uses (A.1.1) to compute the total number of possible permutations. Let `n` be the length of `a`. Let $\mathtt{m}_1 \ldots \mathtt{m}_k$ be the "multiplicities" of `a`, which is an array of counts of each distinct value. For our example, $\mathtt{m}_1 = 2$, $\mathtt{m}_2 = 1$, and $\mathtt{m}_3 = 1$, because `0` appears twice, `2` appears once, and `3` appears once. `k` is the number of distinct values in `a`, which is 3 in our example. Once these variables are assigned, we can simply apply (A.1.1) and return its value.

The actual values in `a`, e.g., `0`, `2`, and `3`, are irrelevant; only their counts are needed. `0`, `2`, and `3` could be replaced with any other three values and this would not change the number of possible permutations. Likewise, the ordering of these values, and the ordering of $\mathtt{m}_1$, $\mathtt{m}_2$, and $\mathtt{m}_3$, are also irrelevant.

[1] `https://en.wikipedia.org/wiki/Permutation#Permutations_of_multisets`

A.2 Permutation rank of a multiset

Algorithm 4.7.2C involves an esoteric formula for computing the rank of a permutation of a multiset, which we label `PERM-RANK-M`. The formula is given by Knuth in the answer to Exercise 4 in Section 7.2.1.2 of [74]. It is difficult to find a pure statement of the formula in a published work and it is often delegated to an exercise for the reader as in page 65 of [80]. This is surprising because the formula is not obvious. Luckily, Knuth provides answers for his exercises.

We do not reproduce the formula here since it has many terms that are themselves complex and require further explanation. We direct the reader to the aforementioned Section 7.2 of [74] or our implementation, with commented C code, given in: `https://github.com/cupslab/guess-calculator-framework/blob/v1.0.0/binaries/pattern_manager.cpp#L599-L780`. The few published papers in this arena seem to focus on permutations of sets [105], rather than multisets, and those algorithms cannot be applied to this problem.

A.3 Converting a mixed-radix number to a single radix

We define the function `SINGLE-RADIX(r)` which takes a mixed-radix number and returns a single number that represents the same quantity. For example, the mixed-radix number:

$$\begin{array}{ll} \mathtt{r_d}: & \langle \quad 2 \quad 3 \quad \rangle \\ \mathtt{r_r}: & [\quad 4 \quad 26 \quad] \end{array}$$

represents the number $2 * 26 + 3 = 55$. We use the subscripts $_{\mathtt{d}}$ and $_{\mathtt{r}}$ to label the digits and radices of the number respectively. The algorithm for computing `SINGLE-RADIX` is given below.

Algorithm R (*Convert mixed-radix number to single radix*). Given a mixed-radix number $\mathtt{r}$ with digits $\mathtt{r_d}$ and radices $\mathtt{r_r}$, compute a single-radix quantity.

Let $\mathtt{n}$ be the length of $\mathtt{r}$: $\mathtt{n} = |\mathtt{r_d}| = |\mathtt{r_r}|$. Let $\mathtt{r}$ be indexed such that $\mathtt{r_d[n-1]}$ is the most significant digit of $\mathtt{r}$ and $\mathtt{r_d[0]}$ is the least significant.

R1. [Initialize.] Set $\mathtt{i} \leftarrow \mathtt{n} - 1$. Set $\mathtt{x} \leftarrow 0$. ($\mathtt{i}$ will maintain our position in the mixed-radix number, and $\mathtt{x}$ will store our final result.)

R2. [Count current radix.] Set $\mathtt{x} \leftarrow \mathtt{x} \cdot \mathtt{r_r[i]}$.

R3. [Count current digit.] Set $\mathtt{x} \leftarrow \mathtt{x} + \mathtt{r_d[i]}$.

R4. [Repeat.] Set $\mathtt{i} \leftarrow \mathtt{i} - 1$. If $\mathtt{i} < 0$, then return $\mathtt{x}$, otherwise go to step R2. ∎

Appendix B

Experiment Configurations

Producing the results in this thesis involved running the guess-calculator framework with many different sets of inputs. This appendix contains tables which describe these inputs, ordered by experiment number. The experiment numbers are roughly in chronological order, though we ran some experiments concurrently with other experiments.

The organization of this appendix is as follows. Each experiment receives its own section, e.g., Experiment 1 is in Appendix B.5. Each *run* within an experiment receives its own page. In many cases, runs correspond to conditions within an experiment. For example, Experiment 2 requires two runs, shown on page 186 and the following page (ignore the details of the tables for now, they will be described later.) The variation between the two conditions is a single learning parameter that controls unseen terminal generation. This is printed under the "Generate unseen terminals?" column. The conditions are plotted together in Figure 5.3 on page 92.

It is also possible to have a single run evaluate multiple conditions. Experiment 8A on page 222 has a single run that evaluates three different test datasets: RockYou, Yahoo! Voices, and CSDN. We produced Figure 7.2(a) on page 140 from this experiment, which plots all three conditions on the same graph. We can evaluate multiple conditions with a single run only when training data and learning parameters do not vary between conditions.

CMU guessability experiments The results from Section 7.4 were produced by an earlier version of the guess-calculator framework that did not include any of the improvements from Chapter 5. While the tables in appendices B.5 through B.24 were generated by a program given configuration files as input, the tables for the CMU guessability experiments (which start at Appendix B.26) were entered manually. The configuration file format for these experiments is now obsolete and is sufficiently different from the current format that it was not worth developing a

table generator. Some runs from these experiments have multiple "Training Data" tables, which means that each table was processed independently and then the resulting corpora were concatenated together. This was used to produce complex, arbitrary weightings among training sources. Unfortunately these weightings were never empirically justified, so we do not recommend this approach in the future.

Columns in each configuration table are described in Appendices B.1, B.2, and B.3. The datasets referenced in the tables are described in Appendix B.4. If any column in a table is blank, then that parameter was not set in the framework. For example, if the "Filter" column is blank for a dataset, then that dataset was not filtered prior to inclusion as an input or output. Some columns are Boolean with value "Y" if the parameter was set in the framework and blank if the parameter is false.

B.1 Test data columns

Display name
The label used for the guessing curve associated with this dataset. If this guessing curve appears in multiple graphs with different labels, the labels are separated by forward slashes.

Name
The name of the dataset.

Source
MTurk, for datasets collected from Mechanical Turk using the methodology of Section 2.2; *public*, for datasets that are freely distributable; *leaked*, for leaked password sets; *paid*, for datasets that we paid for; or *private*, for datasets that we obtained access to in confidence from private organizations.

Filter
A policy. If a filter was applied to this dataset, then only passwords that comply with the named policy are included. Policies are identified using the abbreviated names described in Section 2.4.4.

Random sample?
Was a random sample taken from the dataset? This is distinguished from using the whole dataset after filtering. Random samples are drawn deterministically so that the same sample is drawn on every run. This aids reproducibility.

Size
The final size of the test dataset, after filtering and sampling if applicable.

B.2 Training data columns

Structure source?

Was this training source added to the "Structures" corpus? As shown in Figure 4.2, the guess-calculator framework decouples the "structures" (nonterminals) of the PCFG from its terminals.

Terminal source?

Was this training source added to the "Terminals" corpus? Note that structure sources are always added to both structure and terminal corpora to prevent the generation of an invalid PCFG.[1]

Name

The name of the dataset.

Source

See Section B.1 for a description of this column.

Weight

The weight applied to this source before adding it to the Structures and/or Terminals corpora.

Filter

If a filter was applied to this dataset, then only strings that comply with the named policy are included. Policies are identified using either the abbreviated names described in Section 2.4.4 or with the strings "alpha" or "digsym." The "alpha" filter extracts all contiguous alphabetic substrings from its input and discards the remainder. For example, if an input line were "abc123!de," the filter would extract the separate strings "abc" and "de." The "digsym" filter operates in the same manner, but extracts all contiguous digit or symbol strings. For example, given "abc123!de," the filter would extract "123" and "!" separately.

Use remainder from test?

If a random sample was drawn from this dataset to produce a test set, then this parameter signifies that only the remainder of the dataset, after filtering and sampling, will be used for training. This remainder can be further filtered and sampled to produce a training set.

Random sample?

If a random sample was taken from this dataset, this is the size of the sample, after filtering if applicable.

[1] Structure sources inform the nonterminals of the PCFG, so a nonterminal with no corresponding terminal is invalid.

B.3 Learning parameters

Tokenizer

The tokenizer or tokenizers used to parse structures and terminals from the input data. Tokenization is discussed in Section 5.5.

Probability cutoff

The probability cutoff for the guess-calculator framework. This parameter is required for the intelligent skipping algorithm, presented in Section 4.4.

Generate unseen terminals?

This feature is described in Section 5.3.

Hybrid structures?

This feature is described in Section 5.5.1.

Ignore alphabetic string frequencies?

This parameter causes the framework to ignore string frequencies. It is used for comparison with older methods only, such as Weir et al. circa 2009 [146].

B.4 Datasets

We categorize our input datasets by source: MTurk, public, leaked, paid, and private. These categories are described in Appendix B.1. For each category, we provide statistics for input datasets that can be used by researchers to confirm that they have obtained the correct dataset.

B.4.1 MTurk

The MTurk datasets were collected using the methodology described in Section 2.2 and are named based on the policy used in collecting them. Policy names are described in Section 2.4.4.

MTurk datasets are available for research purposes by contacting the author.

Name	Lines	MD5
basic8	3062	557fe9a1f11d65819de55e49cffa2e06
basic16	2054	9a2ab4d5c470dba1ad69e5370bff0b1f
3class12‡	2774	9dbbbed667c7608e4f3012daba50ec90
comp8	4239	a38b472057bb1147dcf72627ba73b0ef
MTcomp8	3000	93c63d7f0cd2a51422427df779a32c74
MTandrew8	3000	47040e1ce5c5f237a6369154d8f6fb4a

‡The file for this dataset is incorrectly named 3class12_2773.txt and experiments were designed under the assumption that the file contained 2,773 passwords. For example, Experiment 7B randomly selected 2,000 passwords for training and 773 passwords for testing, when the remaining 774 passwords should have been tested. The error was with use of the `wc` command. The file did not end with a newline on the last line, so `wc` only reported 2,773 lines.

B.4.2 Public

Public datasets, excluding the Google datasets, are available at `https://cups.cs.cmu.edu/chili/projects/passwords-gcf-thesis-configs/repository/show/standard%20input%20files`. This location also contains scripts used in cleaning and processing these datasets prior to use with the guess-calculator framework. All files were run through the program `process_wordfreq.py`, which compacts each file before being used in the framework, as a final step. This saves time when a file is used in many runs.

Though the Google Web Corpus is freely available to educational institutions that are members of the Linguistic Data Consortium (LDC), their license does not allow us to distribute it, or derived datasets, publicly.[2] We did not realize this until after all of the experiments for this thesis were completed.

Webster's 2nd

Webster's new international dictionary of the English language, Second Edition, from 1934 [107]. This is a public-domain English-language dictionary. It was processed to extract and lowercase all alphabetic strings.[3]

Lines: 234,936

MD5 of `web2.gz`: 982e63185ce1f22261dec1c016e08f5b

Infl. dictionary

Automatically Generated Inflection Database [1], developed for use with the GNU Aspell spell checker (`http://aspell.net/`). The version used here was developed in 2003. The original source can be download from `http://sourceforge.net/projects/wordlist/files/AGID/Rev%204/agid-4.tar.gz/download`. It was processed in the same manner as Webster's 2nd.

Lines: 252,337

MD5 of `inflection_dict.gz`: 0c0e24af86a9d5bcdb8a56de1ebc11e9

[2] `https://catalog.ldc.upenn.edu/license/web-1t-5-gram-version-1.pdf`

[3] This is accomplished with `egrep -o '[a-zA-Z]+' | tr '[A-Z]' '[a-z]'`.

Google unigrams RY

The unigrams from the Google Web Corpus, found in the file `web_5gram_1/data/1gms/vocab.gz`. The original file was filtered to remove any non-alphabetic tokens. Precise commands are provided below.

```
$ gunzip -c vocab.gz | perl -CSD -nE‘print lc if /^[a-zA-Z]+\t/’ | gzip -
    c > temp.gz
$ ./process_wordfreq.py -n G -w 5.555284392621586e-05 -g temp.gz >
    google_alphatokens_ryeq.gz
```

Since there are over 500 billion tokens covered by this data source, including it without modification would cause it to drown out all other data sources. Therefore, the frequencies in this source have been reduced so that its total weight is equivalent to the total weight of the RockYou dataset, which has approximately 32 million lines. This corresponds to the `-w 5.555284392621586e-05` parameter in the code above which multiplies the weight of this source by $\approx 5.6 \times 10^{-5}$. This choice is rather arbitrary but it gets this source to within an order of magnitude of the other sources. The weight is further reduced or increased by the "Weight" configuration parameter in the framework, given in the tables that follow.

Lines: 4,926,386

MD5 of `google_alphatokens_ryeq.gz`: b0bef15465e9abb20aefa87bc1b585eb

Google n-grams RY

Alphabetic n-grams from the Google Web Corpus, with spaces removed, up to 20 characters in length. As with the Google unigrams, the frequencies in this source were reduced so that its total weight is equivalent to the total weight of the RockYou dataset.

```
# First, gunzip and cat ‘vocab.gz’, ‘2gm*.gz’, ‘3gm*.gz’, etc. into a
    single ‘allstrings.txt’ file.
$ cat allstrings.txt | perl -CSD -nE‘print lc s/[ ]//rg if /^[a-zA-Z ]+\t
    /’ | perl -nE‘print if /^.{1,20}\t/’ | gzip -c > temp.gz
$ ./process_wordfreq.py -n G -w 2.352183e-05 -g temp.gz >
    summed_alphagrams_upto20_ryeq.gz
```

Lines: 823,775,338

MD5 of `summed_alphagrams_upto20_ryeq.gz`: ebffebde55d42d6c95da47ebe253435c

B.4.3 Leaked

We do not provide a copy of the leaked datasets used in this thesis. Though we think it is ethical to use these datasets for research purposes, we do not want to be

involved in distributing them. However, those who wish to reproduce the results of this thesis will need to obtain these datasets for use as training data.

To assist in reproducibility, we provide statistics on these datasets, along with a script we use to clean these datasets prior to use. The script simply removes lines with invalid UTF-8 encodings. The script can be downloaded from `https://cups.cs.cmu.edu/chili/projects/passwords-gcf-thesis-configs/repository/raw/standard%20input%20files/remove_invalid_utf8.pl`. For each file, we provide statistics about the file before and after cleaning.

RockYou

Passwords from RockYou [139].

Lines before cleaning: 32,603,388

Size before cleaning: 289,836,298 bytes

MD5 before cleaning: 188227b3e210bcc36c63bf9ac09b352a

Lines removed: 243

Lines after cleaning: 32,603,145

Size after cleaning: 289,833,959 bytes

MD5 after cleaning: 65978658e4bc3360867f4a250a53682d

Yahoo! Voices

Passwords from the Yahoo! Voices service [14].

Lines before cleaning: 453,488

Size before cleaning: 4,107,423 bytes

MD5 before cleaning: c82f097842187b8d49613d5778531b7b

Lines removed: 0

CSDN

Passwords from the China Software Developer Network [118].

Lines before cleaning: 6,428,632

Size before cleaning: 67,227,626 bytes

MD5 before cleaning: fcd2a818cf7338a1391c146b376a5d0d

Lines removed: 347

Lines after cleaning: 6,428,285

Size after cleaning: 67,221,263 bytes

MD5 after cleaning: b309aeb574599a24dad5198789251dbb

MySpace

This dataset consists of passwords obtained via a phishing attack on users of `myspace.com` [125].

Lines before cleaning: 47,244

Size before cleaning: 418,603 bytes

MD5 before cleaning: 96a30006328ae1af70a9fd0435360b98

Lines removed: 0

Stratfor

Passwords from Strategic Forecasting, Inc. [113]. This dataset consists of passwords that were leaked only after being cracked, making them much weaker than the other datasets.

Lines before cleaning: 804,041

Size before cleaning: 7,676,770 bytes

MD5 before cleaning: 9e809b0f65c3b0fd8453d968dff873bf

Lines removed: 7

Lines after cleaning: 804,034

Size after cleaning: 7,676,718 bytes

MD5 after cleaning: df7b4a4c1d88471ed4e7c374fb9a10a8

Gawker

Passwords used at `gawker.com` [69]. This dataset also consists of passwords that were leaked only after being cracked.

Lines before cleaning: 694,076

Size before cleaning: 5,703,443 bytes

MD5 before cleaning: 48e8a79edc1b47c264b3434c95cd2a42

Lines removed: 12

Lines after cleaning: 694,064

Size after cleaning: 5,703,359 bytes

MD5 after cleaning: b083c5c589b72ad29fc7949136f86f8e

B.4.4 Paid

Only one paid dataset was used, the Openwall dataset [109]. This dataset is meant to be used as a "cracking dictionary." It contains common passwords that have been modified or "mangled" [93] in many different ways.

We do not recommend this dataset. It seems to be a poor choice of data for complex passwords (see Section 6.2.1), even though it contains many complex passwords. Inspection of the dataset indicates that it was generated using a naïve application of a large number of mangling rules, without regard to empirical probability. This means that the distribution of complex passwords in the dataset is not representative of complex passwords in practice. It is also a poor choice in terms of reproducibility—the version of the dataset that we used does not appear to be available for download, because it has been updated since.

In addition to removing invalid UTF-8 with the script mentioned in Section B.4.3, we ignored the first 25 lines of the file, which consist of comments. We also removed all passwords with spaces from the dataset. This was done because we planned to evaluate a policy that did not allow spaces, and we never rebuilt the file when evaluating policies that allow spaces. The commands we use to build the dataset follow.

```
$ tail -n +25 openwall.txt | grep -v ' ' | perl remove_invalid_utf8.pl >
    cleaned_openwall.txt
```

Statistics for this dataset are provided below.

Openwall

Lines before cleaning: 40,532,676

Size before cleaning: 457,210,158 bytes

MD5 before cleaning: 53a645e34a0b74b1a74be10247925792

Lines removed: 683,553

Lines after cleaning: 39,849,123

Size after cleaning: 449,203,512 bytes

MD5 after cleaning: 9c28c374c301ca24969f4360f28ae58c

B.4.5 Private

The CMU guessability experiments in Sections B.26, B.27, and B.28 involve two private datasets. These datasets were only available to us on machines that we could not access. The datasets were also subsequently destroyed and we did

not collect file sizes of MD5 hashes for them. We did collect statistics on their passwords, however, and these statistics were used in the analysis of Section 7.4.2.

Both datasets were created under the `andrew8` policy.

CMUactive
: Passwords from active CMU user accounts, where *active* means that the user had a relationship with the University at the time of data collection.

  Passwords: 25,459

CMUinactive
: Passwords for *inactive* CMU accounts, which were retained for archival purposes.

  Passwords: 17,104

B.5 Experiment 1

Test Data

Display name	Name	Source	Filter	Random sample?	Size
basic8	basic8	MTurk		Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic8		
Y	Y	Yahoo! Voices	leaked	1.0	basic8		
	Y	RockYou	leaked	1.0	not basic8		
	Y	Yahoo! Voices	leaked	1.0	not basic8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-16}	Y		

B.6 Experiment 2

Test Data

Display name	Name	Source	Filter	Random sample?	Size
basic8	basic8	MTurk		Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic8		
Y	Y	Yahoo! Voices	leaked	1.0	basic8		
	Y	RockYou	leaked	1.0	not basic8		
	Y	Yahoo! Voices	leaked	1.0	not basic8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-17}			

Test Data

Display name	Name	Source	Filter	Random sample?	Size
basic8ut	basic8	MTurk		Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic8		
Y	Y	Yahoo! Voices	leaked	1.0	basic8		
	Y	RockYou	leaked	1.0	not basic8		
	Y	Yahoo! Voices	leaked	1.0	not basic8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-17}	Y		

B.7 Experiment 3A

Test Data

Display name	Name	Source	Filter	Random sample?	Size
Weir structures	basic8	MTurk		Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic8		
Y	Y	Yahoo! Voices	leaked	1.0	basic8		
	Y	RockYou	leaked	1.0	not basic8		
	Y	Yahoo! Voices	leaked	1.0	not basic8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-16}	Y		

Test Data

Display name	Name	Source	Filter	Random sample?	Size
Hybrid	basic8	MTurk		Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic8		
Y	Y	Yahoo! Voices	leaked	1.0	basic8		
	Y	RockYou	leaked	1.0	not basic8		
	Y	Yahoo! Voices	leaked	1.0	not basic8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-16}	Y	Y	

B.8 Experiment 3B

Test Data

Display name	Name	Source	Filter	Random sample?	Size
Weir structures / Weir only	RockYou	leaked	4class8	Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	4class8	Y	
Y	Y	Yahoo! Voices	leaked	1.0	4class8		
	Y	RockYou	leaked	1.0	not 4class8	Y	
	Y	Yahoo! Voices	leaked	1.0	not 4class8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-14}	Y		

Test Data

Display name	Name	Source	Filter	Random sample?	Size
Hybrid / Weir hybrid	RockYou	leaked	4class8	Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	4class8	Y	
Y	Y	Yahoo! Voices	leaked	1.0	4class8		
	Y	RockYou	leaked	1.0	not 4class8	Y	
	Y	Yahoo! Voices	leaked	1.0	not 4class8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-14}	Y	Y	

B.9 Experiment 3C

Test Data

Display name	Name	Source	Filter	Random sample?	Size
Policy A	comp8	MTurk			4239

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	comp8		
Y	Y	Yahoo! Voices	leaked	1.0	comp8		
	Y	RockYou	leaked	1.0	not comp8		
	Y	Yahoo! Voices	leaked	1.0	not comp8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	5e-15	Y		

Test Data

Display name	Name	Source	Filter	Random sample?	Size
Policy B	comp8	MTurk			4239

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	comp8		
Y	Y	Yahoo! Voices	leaked	1.0	comp8		
	Y	RockYou	leaked	1.0	not comp8		
	Y	Yahoo! Voices	leaked	1.0	not comp8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	5e-15	Y	Y	

B.10 Experiment 4

Test Data

Display name	Name	Source	Filter	Random sample?	Size
Weir tokenization / SF+U	basic16	MTurk			2054

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic16		
Y	Y	Yahoo! Voices	leaked	1.0	basic16		
	Y	RockYou	leaked	1.0	not basic16		
	Y	Yahoo! Voices	leaked	1.0	not basic16		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}	Y		

Test Data

Display name	Name	Source	Filter	Random sample?	Size
Linguistic tokenization	basic16	MTurk		Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic16		
Y	Y	Yahoo! Voices	leaked	1.0	basic16		
	Y	RockYou	leaked	1.0	not basic16		
	Y	Yahoo! Voices	leaked	1.0	not basic16		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
linguistic	10^{-15}	Y		

Test Data

Display name	Name	Source	Filter	Random sample?	Size
Weir 2009	basic16	MTurk		Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic16		
Y	Y	Yahoo! Voices	leaked	1.0	basic16		
	Y	RockYou	leaked	1.0	not basic16		
	Y	Yahoo! Voices	leaked	1.0	not basic16		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}			Y

Test Data

Display name	Name	Source	Filter	Random sample?	Size
SF	basic16	MTurk		Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic16		
Y	Y	Yahoo! Voices	leaked	1.0	basic16		
	Y	RockYou	leaked	1.0	not basic16		
	Y	Yahoo! Voices	leaked	1.0	not basic16		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}			

Test Data

Display name	Name	Source	Filter	Random sample?	Size
SF+LT	basic16	MTurk			2054

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic16		
Y	Y	Yahoo! Voices	leaked	1.0	basic16		
	Y	RockYou	leaked	1.0	not basic16		
	Y	Yahoo! Voices	leaked	1.0	not basic16		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
linguistic	10^{-15}			

Test Data

Display name	Name	Source	Filter	Random sample?	Size
SF+LT+H	basic16	MTurk			2054

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic16		
Y	Y	Yahoo! Voices	leaked	1.0	basic16		
	Y	RockYou	leaked	1.0	not basic16		
	Y	Yahoo! Voices	leaked	1.0	not basic16		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
linguistic	10^{-15}		Y	

B.11 Experiment 5A

Test Data

Display name	Name	Source	Filter	Random sample?	Size
Kelley et al.	basic8	MTurk		Y	3062

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	Openwall	paid	1.0	basic8		
Y	Y	MySpace	leaked	1P	basic8		
	Y	RockYou	leaked	1.0			
	Y	Openwall	paid	1.0	not basic8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}			Y

Test Data

Display name	Name	Source	Filter	Random sample?	Size
Public only	basic8	MTurk		Y	3062

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic8		
Y	Y	Yahoo! Voices	leaked	1.0	basic8		
	Y	RockYou	leaked	1.0	not basic8		
	Y	Yahoo! Voices	leaked	1.0	not basic8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}			

Test Data

Display name	Name	Source	Filter	Random sample?	Size
+500	basic8	MTurk		Y	2562

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic8		
Y	Y	Yahoo! Voices	leaked	1.0	basic8		
Y	Y	basic8	MTurk	1P		Y	
	Y	RockYou	leaked	1.0	not basic8		
	Y	Yahoo! Voices	leaked	1.0	not basic8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}			

Test Data

Display name	Name	Source	Filter	Random sample?	Size
+1000	basic8	MTurk		Y	2062

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic8		
Y	Y	Yahoo! Voices	leaked	1.0	basic8		
Y	Y	basic8	MTurk	1P		Y	
	Y	RockYou	leaked	1.0	not basic8		
	Y	Yahoo! Voices	leaked	1.0	not basic8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}			

Test Data

Display name	Name	Source	Filter	Random sample?	Size
+1500	basic8	MTurk		Y	1562

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic8		
Y	Y	Yahoo! Voices	leaked	1.0	basic8		
Y	Y	basic8	MTurk	1P		Y	
	Y	RockYou	leaked	1.0	not basic8		
	Y	Yahoo! Voices	leaked	1.0	not basic8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}			

Test Data

Display name	Name	Source	Filter	Random sample?	Size
+2000	basic8	MTurk		Y	1062

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic8		
Y	Y	Yahoo! Voices	leaked	1.0	basic8		
Y	Y	basic8	MTurk	1P		Y	
	Y	RockYou	leaked	1.0	not basic8		
	Y	Yahoo! Voices	leaked	1.0	not basic8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}			

Test Data

Display name	Name	Source	Filter	Random sample?	Size
+2500	basic8	MTurk		Y	562

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic8		
Y	Y	Yahoo! Voices	leaked	1.0	basic8		
Y	Y	basic8	MTurk	1P		Y	
	Y	RockYou	leaked	1.0	not basic8		
	Y	Yahoo! Voices	leaked	1.0	not basic8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}			

B.12 Experiment 5B

Test Data

Display name	Name	Source	Filter	Random sample?	Size
Kelley et al.	comp8	MTurk		Y	4239

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	Openwall	paid	1.0	comp8		
Y	Y	MySpace	leaked	1P	comp8		
	Y	RockYou	leaked	1.0			
	Y	Openwall	paid	1.0	not comp8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-14}			Y

Test Data

Display name	Name	Source	Filter	Random sample?	Size
Public only	comp8	MTurk		Y	4239

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	comp8		
Y	Y	Yahoo! Voices	leaked	1.0	comp8		
	Y	RockYou	leaked	1.0	not comp8		
	Y	Yahoo! Voices	leaked	1.0	not comp8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-14}			

Test Data

Display name	Name	Source	Filter	Random sample?	Size
+500	comp8	MTurk		Y	3739

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	comp8		
Y	Y	Yahoo! Voices	leaked	1.0	comp8		
Y	Y	comp8	MTurk	1P		Y	
	Y	RockYou	leaked	1.0	not comp8		
	Y	Yahoo! Voices	leaked	1.0	not comp8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-14}			

Test Data

Display name	Name	Source	Filter	Random sample?	Size
+1000	comp8	MTurk		Y	3239

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	comp8		
Y	Y	Yahoo! Voices	leaked	1.0	comp8		
Y	Y	comp8	MTurk	1P		Y	
	Y	RockYou	leaked	1.0	not comp8		
	Y	Yahoo! Voices	leaked	1.0	not comp8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-14}			

Test Data

Display name	Name	Source	Filter	Random sample?	Size
+1500	comp8	MTurk		Y	2739

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	comp8		
Y	Y	Yahoo! Voices	leaked	1.0	comp8		
Y	Y	comp8	MTurk	1P		Y	
	Y	RockYou	leaked	1.0	not comp8		
	Y	Yahoo! Voices	leaked	1.0	not comp8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-14}			

Test Data

Display name	Name	Source	Filter	Random sample?	Size
+2000	comp8	MTurk		Y	2239

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	comp8		
Y	Y	Yahoo! Voices	leaked	1.0	comp8		
Y	Y	comp8	MTurk	1P		Y	
	Y	RockYou	leaked	1.0	not comp8		
	Y	Yahoo! Voices	leaked	1.0	not comp8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-14}			

Test Data

Display name	Name	Source	Filter	Random sample?	Size
+2500	comp8	MTurk		Y	1739

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	comp8		
Y	Y	Yahoo! Voices	leaked	1.0	comp8		
Y	Y	comp8	MTurk	1P		Y	
	Y	RockYou	leaked	1.0	not comp8		
	Y	Yahoo! Voices	leaked	1.0	not comp8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-14}			

B.13 Experiment 6

Test Data

Display name	Name	Source	Filter	Random sample?	Size
0	3class12	MTurk		Y	773

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	3class12		
Y	Y	Yahoo! Voices	leaked	1.0	3class12		
	Y	RockYou	leaked	1.0	not 3class12		
	Y	Yahoo! Voices	leaked	1.0	not 3class12		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}	Y	Y	

Test Data

Display name	Name	Source	Filter	Random sample?	Size
100S	3class12	MTurk		Y	773

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	3class12		
Y	Y	Yahoo! Voices	leaked	1.0	3class12		
Y	Y	3class12	MTurk	100.0		Y	2000
	Y	RockYou	leaked	1.0	not 3class12		
	Y	Yahoo! Voices	leaked	1.0	not 3class12		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}	Y	Y	

Test Data

Display name	Name	Source	Filter	Random sample?	Size
100P	3class12	MTurk		Y	773

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	3class12		
Y	Y	Yahoo! Voices	leaked	1.0	3class12		
Y	Y	3class12	MTurk	100P		Y	2000
	Y	RockYou	leaked	1.0	not 3class12		
	Y	Yahoo! Voices	leaked	1.0	not 3class12		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}	Y	Y	

Test Data

Display name	Name	Source	Filter	Random sample?	Size
10P	3class12	MTurk		Y	773

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	3class12		
Y	Y	Yahoo! Voices	leaked	1.0	3class12		
Y	Y	3class12	MTurk	10P		Y	2000
	Y	RockYou	leaked	1.0	not 3class12		
	Y	Yahoo! Voices	leaked	1.0	not 3class12		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}	Y	Y	

Test Data

Display name	Name	Source	Filter	Random sample?	Size
1S	3class12	MTurk		Y	773

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	3class12		
Y	Y	Yahoo! Voices	leaked	1.0	3class12		
Y	Y	3class12	MTurk	1.0		Y	2000
	Y	RockYou	leaked	1.0	not 3class12		
	Y	Yahoo! Voices	leaked	1.0	not 3class12		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}	Y	Y	

Test Data

Display name	Name	Source	Filter	Random sample?	Size
1P	3class12	MTurk		Y	773

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	3class12		
Y	Y	Yahoo! Voices	leaked	1.0	3class12		
Y	Y	3class12	MTurk	1P		Y	2000
	Y	RockYou	leaked	1.0	not 3class12		
	Y	Yahoo! Voices	leaked	1.0	not 3class12		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}	Y	Y	

B.14 Experiment 7A

Test Data

Display name	Name	Source	Filter	Random sample?	Size
MTurk	3class12	MTurk		Y	773
RockYou	RockYou	leaked	3class12	Y	1000
Yahoo	Yahoo! Voices	leaked	3class12	Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	3class12	Y	
Y	Y	Yahoo! Voices	leaked	1.0	3class12	Y	
	Y	RockYou	leaked	1.0	not 3class12		
	Y	Yahoo! Voices	leaked	1.0	not 3class12		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}	Y	Y	

B.15 Experiment 7B

Test Data

Display name	Name	Source	Filter	Random sample?	Size
MTurk	3class12	MTurk		Y	773
RockYou	RockYou	leaked	3class12	Y	1000
Yahoo	Yahoo! Voices	leaked	3class12	Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	3class12	Y	
Y	Y	Yahoo! Voices	leaked	1.0	3class12	Y	
Y	Y	3class12	MTurk	10P		Y	2000
	Y	RockYou	leaked	1.0	not 3class12		
	Y	Yahoo! Voices	leaked	1.0	not 3class12		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}	Y	Y	

B.16 Experiment 8A

Test Data

Display name	Name	Source	Filter	Random sample?	Size
RockYou	RockYou	leaked	basic6	Y	1000
Yahoo	Yahoo! Voices	leaked	basic6	Y	1000
CSDN	CSDN	leaked	basic6	Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic6	Y	
	Y	RockYou	leaked	1.0	not basic6	Y	

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}			Y

B.17 Experiment 8B

Test Data

Display name	Name	Source	Filter	Random sample?	Size
RockYou	RockYou	leaked	basic6	Y	1000
Yahoo / YahoowT	Yahoo! Voices	leaked	basic6	Y	1000
CSDN / CSDNwT	CSDN	leaked	basic6	Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic6	Y	
Y	Y	Yahoo! Voices	leaked	1.0	basic6	Y	
Y	Y	CSDN	leaked	1.0	basic6	Y	
	Y	RockYou	leaked	1.0	not basic6	Y	
	Y	Yahoo! Voices	leaked	1.0	not basic6		
	Y	CSDN	leaked	1.0	not basic6		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}			Y

B.18 Experiment 8C

Test Data

Display name	Name	Source	Filter	Random sample?	Size
RockYou	RockYou	leaked	basic6	Y	1000
Yahoo / YahoowI	Yahoo! Voices	leaked	basic6	Y	1000
CSDN / CSDNwI	CSDN	leaked	basic6	Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic6	Y	
	Y	RockYou	leaked	1.0	not basic6	Y	

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}	Y	Y	

B.19 Experiment 8D

Test Data

Display name	Name	Source	Filter	Random sample?	Size
RockYou	RockYou	leaked	basic6	Y	1000
Yahoo / YahoowIwT	Yahoo! Voices	leaked	basic6	Y	1000
CSDN / CSDNwIwT	CSDN	leaked	basic6	Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic6	Y	
Y	Y	Yahoo! Voices	leaked	1.0	basic6	Y	
Y	Y	CSDN	leaked	1.0	basic6	Y	
	Y	RockYou	leaked	1.0	not basic6	Y	
	Y	Yahoo! Voices	leaked	1.0	not basic6		
	Y	CSDN	leaked	1.0	not basic6		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}	Y	Y	

B.20 Experiment 8E

Test Data

Display name	Name	Source	Filter	Random sample?	Size
RockYou	RockYou	leaked	basic6	Y	1000
Yahoo	Yahoo! Voices	leaked	basic6	Y	1000
CSDN5000wI	CSDN	leaked	basic6	Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic6	Y	
Y	Y	Yahoo! Voices	leaked	1.0	basic6	Y	
Y	Y	CSDN	leaked	10P	basic6	Y	5000
	Y	RockYou	leaked	1.0	not basic6	Y	
	Y	Yahoo! Voices	leaked	1.0	not basic6		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}	Y	Y	

B.21 Experiment 9A

Test Data

Display name	Name	Source	Filter	Random sample?	Size
Weir 2009	RockYou	leaked	4class8	Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	4class8	Y	
Y	Y	Yahoo! Voices	leaked	1.0	4class8		
	Y	RockYou	leaked	1.0	not 4class8	Y	
	Y	Yahoo! Voices	leaked	1.0	not 4class8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-14}			Y

Test Data

Display name	Name	Source	Filter	Random sample?	Size
Improved	RockYou	leaked	4class8	Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	4class8	Y	
Y	Y	Yahoo! Voices	leaked	1.0	4class8		
	Y	RockYou	leaked	1.0	not 4class8	Y	
	Y	Yahoo! Voices	leaked	1.0	not 4class8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-14}	Y	Y	

B.22 Experiment 9B

Test Data

Display name	Name	Source	Filter	Random sample?	Size
Weir 2009	RockYou	leaked	basic8	Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic8	Y	
Y	Y	Yahoo! Voices	leaked	1.0	basic8		
	Y	RockYou	leaked	1.0	not basic8	Y	
	Y	Yahoo! Voices	leaked	1.0	not basic8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}			Y

Test Data

Display name	Name	Source	Filter	Random sample?	Size
Improved	RockYou	leaked	basic8	Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic8	Y	
Y	Y	Yahoo! Voices	leaked	1.0	basic8		
	Y	RockYou	leaked	1.0	not basic8	Y	
	Y	Yahoo! Voices	leaked	1.0	not basic8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}	Y	Y	

B.23 Experiment 10

Test Data

Display name	Name	Source	Filter	Random sample?	Size
SF+U+GS	basic16	MTurk			2054

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	basic16		
Y	Y	Yahoo! Voices	leaked	1.0	basic16		
	Y	RockYou	leaked	1.0	not basic16		
	Y	Yahoo! Voices	leaked	1.0	not basic16		
	Y	Google n-grams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}	Y		

B.24 Experiment 11

Test Data

Display name	Name	Source	Filter	Random sample?	Size
Unsupervised hybrid	RockYou	leaked	4class8	Y	1000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	4class8	Y	
Y	Y	Yahoo! Voices	leaked	1.0	4class8		
	Y	RockYou	leaked	1.0	not 4class8	Y	
	Y	Yahoo! Voices	leaked	1.0	not 4class8		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
unsupervised (structures) character-class (terminals)	10^{-15}	Y	Y	

B.25 Experiment 12

Test Data

Display name	Name	Source	Filter	Random sample?	Size
Our approach	3class12	MTurk			2774

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y	Y	RockYou	leaked	1.0	3class12		
Y	Y	Yahoo! Voices	leaked	1.0	3class12		
	Y	RockYou	leaked	1.0	not 3class12		
	Y	Yahoo! Voices	leaked	1.0	not 3class12		
	Y	Webster's 2nd	leaked	0.1	alpha		
	Y	Infl. dictionary	leaked	0.1	alpha		
	Y	Google unigrams RY	leaked	0.1	alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-15}	Y	Y	

B.26 Experiment CMU-1

Test Data

Display name	Name	Source	Filter	Random sample?	Size
CMUactive	CMU Andrew	private			25459
MTcomp8	MTcomp8	MTurk			3000
MTsim	MTandrew8	MTurk			3000
RYcomp8	RockYou	leaked	andrew8	Y	1000
Ycomp8	Yahoo! Voices	leaked	andrew8	Y	1000
CSDNcomp8	CSDN	leaked	andrew8	Y	1000
SFcomp8	Stratfor	leaked	andrew8	Y	1000
Gcomp8	Gawker	leaked	andrew8		896

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y		MySpace	leaked	1.0	4class8		
Y		Openwall	paid	1.0	4class8		
Y		RockYou	leaked	1.0	4class8	Y	
Y		Yahoo! Voices	leaked	1.0	4class8	Y	
Y		CSDN	leaked	1.0	4class8	Y	
Y		Stratfor	leaked	1.0	4class8	Y	
Y		Gawker	leaked	1.0	4class8	Y	

Terminals were learned from both of the following lists of sources. The first list specifies sources from which digits and symbols were learned, and alphabetic terminals were learned from the second list. Alphabetic string frequencies are ignored, so weights are irrelevant for alphabetic sources.

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
	Y	MySpace	leaked	1.0	digsym		
	Y	Openwall	paid	1.0	digsym		
	Y	RockYou	leaked	1.0	digsym	Y	
	Y	Yahoo! Voices	leaked	1.0	digsym	Y	
	Y	CSDN	leaked	1.0	digsym	Y	
	Y	Stratfor	leaked	1.0	digsym	Y	
	Y	Gawker	leaked	1.0	digsym	Y	
	Y	MySpace	leaked		alpha		
	Y	Openwall	paid		alpha		
	Y	RockYou	leaked		alpha	Y	
	Y	Yahoo! Voices	leaked		alpha	Y	
	Y	CSDN	leaked		alpha	Y	
	Y	Stratfor	leaked		alpha	Y	
	Y	Gawker	leaked		alpha	Y	
	Y	Webster's 2nd	public		alpha		
	Y	Infl. dictionary	public		alpha		
	Y	Google unigrams RY	public		alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-16}			Y

B.27 Experiment CMU-2

Test Data

Display name	Name	Source	Filter	Random sample?	Size
CMUactive	CMU Andrew	private		Y	5459
MTcomp8	MTcomp8	MTurk			3000
MTsim	MTandrew8	MTurk			3000
RYcomp8	RockYou	leaked	andrew8	Y	1000
Ycomp8	Yahoo! Voices	leaked	andrew8	Y	1000
CSDNcomp8	CSDN	leaked	andrew8	Y	1000
SFcomp8	Stratfor	leaked	andrew8	Y	1000
Gcomp8	Gawker	leaked	andrew8		896

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y		MySpace	leaked	1.0	4class8		
Y		Openwall	paid	1.0	4class8		
Y		RockYou	leaked	1.0	4class8	Y	
Y		Yahoo! Voices	leaked	1.0	4class8	Y	
Y		CSDN	leaked	1.0	4class8	Y	
Y		Stratfor	leaked	1.0	4class8	Y	
Y		Gawker	leaked	1.0	4class8	Y	
Y		CMUactive	private	1P		Y	
Y		CMUinactive	private	1.0	4class8		15000

Terminals were learned from both of the following lists of sources. The first list specifies sources from which digits and symbols were learned, and alphabetic terminals were learned from the second list. 1P weighting (described in Section 6.2.2) applies to preceding sources in the current list only. Alphabetic string frequencies are ignored, so weights are irrelevant for alphabetic sources.

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
	Y	MySpace	leaked	1.0	digsym		
	Y	Openwall	paid	1.0	digsym		
	Y	RockYou	leaked	1.0	digsym	Y	
	Y	Yahoo! Voices	leaked	1.0	digsym	Y	
	Y	CSDN	leaked	1.0	digsym	Y	
	Y	Stratfor	leaked	1.0	digsym	Y	
	Y	Gawker	leaked	1.0	digsym	Y	
	Y	CMUactive	private	1P	digsym	Y	
	Y	CMUinactive	private	1.0	digsym		15000

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
	Y	MySpace	leaked		alpha		
	Y	Openwall	paid		alpha		
	Y	RockYou	leaked		alpha	Y	
	Y	Yahoo! Voices	leaked		alpha	Y	
	Y	CSDN	leaked		alpha	Y	
	Y	Stratfor	leaked		alpha	Y	
	Y	Gawker	leaked		alpha	Y	
	Y	Webster's 2nd	public		alpha		
	Y	Infl. dictionary	public		alpha		
	Y	Google unigrams RY	public		alpha		
	Y	CMUactive	private		alpha	Y	
	Y	CMUinactive	private		alpha		15000

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-16}			Y

B.28 Experiment CMU-3

This experiment consisted of three runs on the same training and test data sources. On each run, the test data was one third of the CMUactive dataset, and the remaining two thirds were used for training. Tables for a single run are provided here.

Test datasets with no display name were drawn randomly from the original datasets, but not used.

Test Data

Display name	Name	Source	Filter	Random sample?	Size
CMUactive	CMU Andrew	private		Y	$\frac{1}{3}$ of 25459
	RockYou	leaked	andrew8	Y	1000
	Yahoo! Voices	leaked	andrew8	Y	1000
	CSDN	leaked	andrew8	Y	1000
	Stratfor	leaked	andrew8	Y	1000
	Gawker	leaked	andrew8		896

1P weighting (described in Section 6.2.2) was used on the combination of CMUinactive and remaining CMUactive passwords. This is denoted by a shared weight entry for both training sources.

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
Y		MySpace	leaked	1.0	4class8		
Y		Openwall	paid	1.0	4class8		
Y		RockYou	leaked	1.0	4class8	Y	
Y		Yahoo! Voices	leaked	1.0	4class8	Y	
Y		CSDN	leaked	1.0	4class8	Y	
Y		Stratfor	leaked	1.0	4class8	Y	
Y		Gawker	leaked	1.0	4class8	Y	
Y		CMUactive	private	1P		Y	
Y		CMUinactive	private		4class8		

Terminals were learned from both of the following lists of sources. The first list specifies sources from which digits and symbols were learned, and alphabetic terminals were learned from the second list. 1P weighting applies to preceding sources in the current list only. Alphabetic string frequencies are ignored, so weights are irrelevant for alphabetic sources.

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
	Y	MySpace	leaked	1.0	digsym		
	Y	Openwall	paid	1.0	digsym		
	Y	RockYou	leaked	1.0	digsym	Y	
	Y	Yahoo! Voices	leaked	1.0	digsym	Y	
	Y	CSDN	leaked	1.0	digsym	Y	
	Y	Stratfor	leaked	1.0	digsym	Y	
	Y	Gawker	leaked	1.0	digsym	Y	
	Y	CMUactive	private	1P	digsym	Y	
	Y	CMUinactive	private		digsym		

Training Data

Structure source?	Terminal source?	Name	Source	Weight	Filter	Use remainder from test?	Random sample?
	Y	MySpace	leaked		alpha		
	Y	Openwall	paid		alpha		
	Y	RockYou	leaked		alpha	Y	
	Y	Yahoo! Voices	leaked		alpha	Y	
	Y	CSDN	leaked		alpha	Y	
	Y	Stratfor	leaked		alpha	Y	
	Y	Gawker	leaked		alpha	Y	
	Y	Webster's 2nd	public		alpha		
	Y	Infl. dictionary	public		alpha		
	Y	Google unigrams RY	public		alpha		
	Y	CMUactive	private		alpha	Y	
	Y	CMUinactive	private		alpha		

Learning parameters

Tokenizer	Probability cutoff	Generate unseen terminals?	Hybrid structures?	Ignore alphabetic string frequencies?
character-class	10^{-16}			Y

Appendix C

Modifications to the Unsupervised Tokenization Algorithm

We contacted Berg-Kirkpatrick et al. to obtain a copy of the code they used in their paper, "Painless unsupervised learning with features" [3], and they graciously shared it with us. The goal of their system is to identify words in unbroken text based purely on an unsupervised learning algorithm. Passwords can exhibit the properties of unbroken text, since most users do not use spaces to separate words.

We made a number of modifications to Berg-Kirkpatrick et al.'s code to allow it to be used within our framework, and these modifications are described here. Though we do not describe the algorithm in detail, we try to provide enough detail to reimplement our modifications.

Inputs and outputs The original implementation read a flat file of strings, one per line. We changed this to conform to our input format which stores strings along with their weight and source identifiers. We also modified the output methods so that tokenized strings were written out in the same standard format used by other tokenizers in the guess-calculator framework.[1] When writing tokenized strings back out, we do so along with the weight and source identifiers from the input.

The original implementation also expected the input to be in IPA (International Phonetic Alphabet) format, which is not applicable to passwords. We modified the input reader to expect UTF-8.

Weighted strings The original implementation did not have support for weighted input strings, which is a core feature of our framework. We changed their implementation to weight examples based on the weight provided in the input

[1] See `https://github.com/cupslab/guess-calculator-framework/blob/v1.0.0/USAGE.md#writing-filters-and-tokenizers`.

corpus. This is used in computing the likelihood function that the algorithm maximizes. We weight the contribution to the likelihood function of each example linearly based on its weight in the input corpus.

Features The Berg-Kirkpatrick et al. paper does not introduce the unsupervised-tokenization algorithm. It was introduced in a previous paper by many of the same authors [85]. Instead, it adds in the idea of using features of the input strings to improve the performance of the algorithm greatly. For example, they use prefixes and suffixes of tokens as features. This means that particular prefixes and suffixes, like "th" or "nd," can be used as signals of word boundaries. Their implementation defines different "feature templates" that can be enabled or disabled as desired.

We follow them and enable the following features:

Indicator This is the basic feature that counts occurrences of a particular substring. We modify this feature to make it case-insensitive.

Length Take the length of a potential token as a signal that it might be a word. This is separate from the penalty applied to tokens based on their length, which is also applied. This was described in Section 5.5.3.1 and the original paper [3].

Prefix/Suffix Use prefixes and suffixes of lengths 2 to 4 as features. We modify this feature to make it case-insensitive.

Shaped prefix/suffix In addition to trying to use the case-insensitive prefixes and suffixes, we also use a phonetic transform of these strings. This is suggested by Berg-Kirkpatrick et al. Since literal strings might not contain enough information to work well as a feature, a simple phonetic transform can work better. For example, the prefix `th` maps to the code `cc`, which indicates two consonants in a row. Such patterns might be good indications of word boundaries.

Unlike Berg-Kirkpatrick et al., our inputs are more diverse than consonants and vowels. Therefore, we also assume that case might be a good indication of word boundaries. Therefore, we record the following classes: uppercase consonants, uppercase vowels, lowercase consonants, lowercase vowels, digits, spaces, and symbols.

Bibliography

[1] Atkinson, K. Automatically generated inflection database (AGID). `http://wordlist.aspell.net/agid-readme/`, 2011.

[2] Ballard, L., Kamara, S., and Reiter, M. K. The practical subtleties of biometric key generation. In *Proceedings of the 17th USENIX Security Symposium* (Berkeley, CA, USA, 2008), 61–74.

[3] Berg-Kirkpatrick, T., Bouchard-Côté, A., DeNero, J., and Klein, D. Painless unsupervised learning with features. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, HLT '10, Association for Computational Linguistics (Stroudsburg, PA, USA, 2010), 582–590.

[4] Biddle, R., Chiasson, S., and van Oorschot, P. C. Graphical passwords: Learning from the first twelve years. *ACM Comput. Surv. 44*, 4 (Sept. 2012), 19:1–19:41.

[5] Blum, A., Kalai, A., and Langford, J. Beating the hold-out: Bounds for k-fold and progressive cross-validation. In *Proceedings of the twelfth annual conference on Computational learning theory* (1999), 203–208.

[6] Bonneau, J. *Guessing human-chosen secrets*. PhD thesis, University of Cambridge, May 2012.

[7] Bonneau, J. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *SP '12: Proceedings of the 2012 IEEE Symposium on Security and Privacy* (2012).

[8] Bonneau, J., Herley, C., van Oorschot, P. C., and Stajano, F. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *SP '12: Proceedings of the 2012 IEEE Symposium on Security and Privacy* (2012).

[9] Bonneau, J., and Preibusch, S. The password thicket: technical and market failures in human authentication on the web. In *WEIS '10: Proceedings of the 9th Workshop on the Economics of Information Security* (2010).

[10] Bonneau, J., and Schechter, S. Towards reliable storage of 56-bit secrets in human memory. In *Proceedings of the 23rd USENIX Security Symposium*, USENIX (August 2014).

[11] Bonneau, J., and Shutova, E. Linguistic properties of multi-word passphrases. In *USEC '12: Workshop on Usable Security* (2012).

[12] Bonneau, J., and Xu, R. Of contraseñas, sysmawt, and mìmă: Character encoding issues for web passwords. In *Web 2.0 Security & Privacy* (May 2012).

[13] Boztas, S. Entropies, guessing, and cryptography. Tech. Rep. 6, Department of Mathematics, Royal Melbourne Institute of Technology, 1999.

[14] Brading, A. Yahoo Voices hacked, nearly half a million emails and passwords stolen. *Naked Security* (July 2012).

[15] Brantz, T., and Franz, A. The google web 1T 5-gram corpus. Tech. Rep. LDC2006T13, Linguistic Data Consortium, 2006.

[16] Brown, R. D. Corpus-driven splitting of compound words. In *Proceedings of the Ninth International Conference on Theoretical and Methodological Issues in Machine Translation* (2002).

[17] Bryant, K., and Campbell, J. User behaviours associated with password security and management. *Australasian Journal of Information Systems 14*, 1 (2006).

[18] Buhrmester, M., Kwang, T., and Gosling, S. D. Amazon's mechanical turk a new source of inexpensive, yet high-quality, data? *Perspectives on Psychological Science 6*, 1 (2011), 3–5.

[19] Burr, W. E., Dodson, D. F., Newton, E. M., Perlner, R. A., Polk, W. T., Gupta, S., and Nabbus, E. A. Electronic authentication guideline. *NIST Special Publication 800-63-1* (2011).

[20] Burr, W. E., Dodson, D. F., Newton, E. M., Perlner, R. A., Polk, W. T., Gupta, S., and Nabbus, E. A. Electronic authentication guideline. *NIST Special Publication 800-63-2* (2013).

[21] Burr, W. E., Dodson, D. F., and Polk, W. T. Electronic authentication guideline. *NIST Special Publication 800-63* (2006).

[22] Campbell, J., and Bryant, K. Password Composition and Security: An Exploratory Study of User Practice. *Proc. ACIS* (2004).

[23] Canty, A., and Ripley, B. D. *boot: Bootstrap R (S-Plus) Functions*, 2014. R package version 1.3-13.

[24] Carlton, A. G. On the bias of information estimates. *Psychological Bulletin 71*, 2 (1969), 108–109.

[25] Carstens, D. S., Malone, L. C., and McCauley-Bell, P. Applying chunking theory in organizational password guidelines. *Journal of Information, Information Technology, and Organizations 1* (2006), 97–113.

[26] Charniak, E. *Statistical Language Learning*. MIT Press, Cambridge, MA, USA, 1994.

[27] Chen, S. F., and Goodman, J. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics*, ACL '96, Association for Computational Linguistics (Stroudsburg, PA, USA, 1996), 310–318.

[28] Chernick, M. R. Resampling methods. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 2*, 3 (2012), 255–262.

[29] Chi, C.-H., Ding, C., and Lim, A. Word segmentation and recognition for web document framework. In *Proceedings of the eighth international conference on Information and knowledge management*, CIKM '99, ACM (New York, NY, USA, 1999), 458–465.

[30] Chiasson, S., Forget, A., Stobert, E., van Oorschot, P. C., and Biddle, R. Multiple password interference in text passwords and click-based graphical passwords. In *CCS '09: Proceedings of the 16th ACM Conference on Computer and Communications Security*, ACM (New York, NY, USA, 2009), 500–511.

[31] Cluley, G. The worst passwords you could ever choose exposed by yahoo voices hack. *Naked Security* (July 2012).

[32] Cluley, G. Evernote hacked - almost 50 million passwords reset after security breach. *Naked Security* (Mar. 2013).

[33] Collins, M. Head-driven statistical models for natural language parsing. *Comput. Linguist. 29*, 4 (Dec. 2003), 589–637.

[34] Conover, W. *Practical Nonparametric Statistics*. Cram101 Series. Cram101 Incorporated, 2006.

[35] Cormack, R. S., and Mantel, N. Fisher's exact test: The marginal totals as seen from two different angles. *Journal of the Royal Statistical Society. Series D (The Statistician) 40*, 1 (Jan. 1991), 27–34. ArticleType: research-article / Full publication date: 1991 / Copyright Âl' 1991 Royal Statistical Society.

[36] Cox, D. R. Regression models and life-tables. *Journal of the Royal Statistical Society. Series B (Methodological) 34*, 2 (1972), 187–220.

[37] Davis, D., Monrose, F., and Reiter, M. K. On user choice in graphical password schemes. In *Proceedings of the 13th USENIX Security Symposium* (2004).

[38] Davison, A. C., and Hinkley, D. V. *Bootstrap Methods and Their Applications*. Cambridge University Press, Cambridge, 1997. ISBN 0-521-57391-2.

[39] Designer, S. John the Ripper. `http://www.openwall.com/john/`, 1996-2010.

[40] Devillers, M. M. A. Analyzing password strength. *Radboud University Nijmegen, Tech. Rep* (2010).

[41] Dhamija, R., Tygar, J. D., and Hearst, M. Why phishing works. In *CHI '06: Proceedings of the 24th ACM SIGCHI Conference on Human Factors in Computing Systems*, ACM (New York, NY, USA, 2006), 581–590.

[42] Dietterich, T. G. Ensemble methods in machine learning. In *Multiple Classifier Systems*, vol. 1857 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2000, 1–15.

[43] Downs, J. S., Holbrook, M. B., Sheng, S., and Cranor, L. F. Are your participants gaming the system?: screening mechanical turk workers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, ACM (New York, NY, USA, 2010), 2399–2402.

[44] Efron, B., and Gong, G. A leisurely look at the bootstrap, the jackknife, and cross-validation. *The American Statistician 37*, 1 (1983), 36–48.

[45] Fahl, S., Harbach, M., Acar, Y., and Smith, M. On the ecological validity of a password study. In *Proceedings of the Ninth Symposium on Usable Privacy and Security*, SOUPS '13, ACM (New York, NY, USA, 2013), 13:1–13:13.

[46] Fienberg, S. E. Bayesian models and methods in public policy and government settings. *Statistical Science 26*, 2 (May 2011), 212–226. Mathematical Reviews number (MathSciNet): MR2858384; Zentralblatt MATH identifier: 06075158.

[47] Florêncio, D., and Herley, C. Where do security policies come from? In *SOUPS '10: Proceedings of the 6th Symposium on Usable Privacy and Security*, ACM (2010).

[48] Florêncio, D., Herley, C., and van Oorschot, P. C. An administrator's guide to internet password research. In *Proceedings of the 28th USENIX Conference on Large Installation System Administration*, LISA'14, USENIX Association (Berkeley, CA, USA, 2014), 35–52.

[49] Florêncio, D., Herley, C., and van Oorschot, P. C. Password portfolios and the finite-effort user: Sustainably managing large numbers of accounts. In *23rd USENIX Security Symposium (USENIX Security 14)*, USENIX Association (San Diego, CA, Aug. 2014), 575–590.

[50] Friedman, J., Hastie, T., and Tibshirani, R. Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software 33*, 1 (2010), 1–22.

[51] Gale, W. A., and Sampson, G. Good-turing frequency estimation without tears. *Journal of Quantitative Linguistics 2*, 3 (1995), 217–237.

[52] Good, I. J. The population frequencies of species and the estimation of population parameters. *Biometrika 40* (1953), 237–264.

[53] Good, P. I. *Permutation, Parametric, and Bootstrap Tests of Hypotheses (Springer Series in Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2004.

[54] Goodin, D. Anatomy of a hack: How crackers ransack passwords like "qeadzcwrsfxv1331". *Ars Technica* (May 2013).

[55] Harrington, D. P., and Fleming, T. R. A Class of Rank Test Procedures for Censored Survival Data. *Biometrika 69*, 3 (1982), pp. 553–566.

[56] Hastie, T., Tibshirani, R., Friedman, J., and Franklin, J. The elements of statistical learning: data mining, inference and prediction. *The Mathematical Intelligencer 27*, 2 (2005), 83–85.

[57] Hayes, B. First links in the markov chain. *American Scientist 101*, 2 (Mar. 2013).

[58] Herley, C., and van Oorschot, P. C. A research agenda acknowledging the persistence of passwords. *IEEE Security & Privacy Magazine* (2012).

[59] Hollander, M., and Wolfe, D. *Nonparametric Statistical Methods*. Wiley Series in Probability and Statistics. Wiley, 1999.

[60] Holm, S. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* (1979), 65–70.

[61] Hunt, T. The science of password selection. `www.troyhunt.com/2011/07/science-of-password-selection.html`, July 2011, retrieved September 2012.

[62] Hwang, J. 93% of top passwords appear in LinkedIn leak. `http://www.johnvey.com/blog/2012/06/93-of-top-passwords-appear-in-linkedin-leak`, June 2012.

[63] Imperva. Consumer password worst practices. `www.imperva.com/docs/WP_Consumer_Password_Worst_Practices.pdf`, 2010, retrieved September 2012.

[64] Jakobsson, M., and Dhiman, M. The benefits of understanding passwords. In *Mobile Authentication*, SpringerBriefs in Computer Science. Springer New York, Jan. 2013, 5–24.

[65] Joshi, A., Shanker, K. V., and Weir, D. The convergence of mildly context-sensitive grammar formalisms. *Technical Reports (CIS)* (Jan. 1990).

[66] Karlof, C., Tygar, J. D., and Wagner, D. Conditioned-safe ceremonies and a user study of an application to web authentication. In *SOUPS '09: Proceedings of the 5th Symposium on Usable Privacy and Security*, ACM (New York, NY, USA, 2009).

[67] Kelley, P. G., Komanduri, S., Mazurek, M. L., Shay, R., Vidas, T., Bauer, L., Christin, N., Cranor, L. F., and Lopez, J. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, Oakland '12, IEEE Computer Society (Washington, DC, USA, 2012), 523–537.

[68] Kelly, S. M. LinkedIn confirms, apologizes for stolen password breach. *Mashable* (June 2012).

[69] Kennedy, D. The real lessons of gawker's security mess. *Forbes* (Dec. 2010).

[70] Kennedy, J. Massive cyber security breach hits sweden. *Silicon Republic* (Oct. 2011).

[71] Kirk, J. Dating site eHarmony confirms password breach. *Computerworld* (June 2012).

[72] Kittur, A., Chi, E. H., and Suh, B. Crowdsourcing user studies with Mechanical Turk. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2008), 453–456.

[73] Knuth, D. E. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[74] Knuth, D. E. *The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations (Art of Computer Programming)*. Addison-Wesley Professional, 2005.

[75] Kohavi, R. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International joint Conference on artificial intelligence*, vol. 14 (1995), 1137–1145.

[76] Komanduri, S., Shay, R., Kelley, P. G., Mazurek, M. L., Bauer, L., Christin, N., Cranor, L. F., and Egelman, S. Of passwords and people: measuring the effect of password-composition policies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, ACM (New York, NY, USA, 2011), 2595–2604.

[77] KoreLogic. "Crack Me If You Can" - DEFCON 2010. `http://contest-2010.korelogic.com/rules.html`, 2010-.

[78] KoreLogic. "Analytical Solutions: Password Recovery Service". `http://contest-2010.korelogic.com/prs.html`, 2015.

[79] Korkmaz, I., and Dalkilic, M. E. The weak and the strong password preferences: a case study on turkish users. In *Proceedings of the 3rd international conference on Security of information and networks*, SIN '10, ACM (New York, NY, USA, 2010), 56–61.

[80] Kreher, D. L., and Stinson, D. R. *Combinatorial algorithms: generation, enumeration, and search*, vol. 7. CRC press, 1998.

[81] Kuo, C., Romanosky, S., and Cranor, L. F. Human selection of mnemonic phrase-based passwords. In *SOUPS '06: Proceedings of the 2nd Symposium on Usable Privacy and Security*, ACM (2006), 67–78.

[82] Kushle, S. Lessons from PlayStation network security breach. http://subinkushle.wordpress.com/2011/05/05/looking-at-playstation-network-security-breach, May 2011.

[83] Li, J., Wang, H., Ren, D., and Li, G. Discriminative pruning of language models for chinese word segmentation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, ACL-44, Association for Computational Linguistics (Stroudsburg, PA, USA, 2006), 1001–1008.

[84] Li, Z., Han, W., and Xu, W. A large-scale empirical analysis of chinese web passwords. In *23rd USENIX Security Symposium (USENIX Security 14)*, USENIX Association (San Diego, CA, Aug. 2014), 559–574.

[85] Liang, P., and Klein, D. Online EM for unsupervised models. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, NAACL '09, Association for Computational Linguistics (Stroudsburg, PA, USA, 2009), 611–619.

[86] Liu, D. C., and Nocedal, J. On the limited memory bfgs method for large scale optimization. *Math. Program. 45*, 3 (Dec. 1989), 503–528.

[87] Lloyd, S. Least squares quantization in pcm. *Information Theory, IEEE Transactions on 28*, 2 (Mar 1982), 129–137.

[88] Ma, J., Yang, W., Luo, M., and Li, N. A study of probabilistic password models. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, IEEE Computer Society (Washington, DC, USA, 2014), 689–704.

[89] Malone, D., and Maher, K. Investigating the distribution of password choices. In *Proc. WWW* (2012).

[90] Mann, H. B., and Whitney, D. R. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[91] Mannan, M., and van Oorschot, P. C. Digital objects as passwords. In *HOTSEC'08: Proceedings of the 3rd Conference on Hot topics in Security*, USENIX Association (Berkeley, CA, USA, 2008), 1–6.

[92] Marechal, S. Advances in password cracking. *Journal in Computer Virology 4*, 1 (2008), 73–81.

[93] Marechal, S. Automatic mangling rules generation. In *Automatic mangling rules generation* (Department of Informatics, University of Oslo, Norway, Dec. 2012).

[94] Max, J. Quantizing for minimum distortion. *Information Theory, IRE Transactions on 6*, 1 (March 1960), 7–12.

[95] Mazurek, M. L., Komanduri, S., Vidas, T., Bauer, L., Christin, N., Cranor, L. F., Kelley, P. G., Shay, R., and Ur, B. Measuring password guessability for an entire university. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, ACM (New York, NY, USA, 2013), 173–186.

[96] McDonald, J. H. *Handbook of Biological Statistics*, 3rd ed. ed. Sparky House Publishing, Baltimore, Maryland, USA, 2014.

[97] Medlin, B. D., Cazier, J. A., and Foulk, D. P. Analyzing the vulnerability of US hospitals to social engineering attacks: how many of your employees would share their password? *IJISP 2*, 3 (2008).

[98] Melicher, W., Kurilova, D., Segreti, S. M., Kalvani, P., Shay, R., Ur, B., Bauer, L., Christin, N., Cranor, L. F., and Mazurek, M. L. Usability and security of text passwords on mobile devices. In *Proceedings of the 2016 Annual ACM Conference on Human Factors in Computing Systems*, CHI '16 (Conditionally accepted), ACM (2016).

[99] Menezes, A., van Oorschot, P. C., and Vanstone, S. *Handbook of Applied Cryptography*. CRC Press, 1997.

[100] Mikolov, T. *Statistical Language Models Based on Neural Networks*. PhD thesis, Ph. D. thesis, Brno University of Technology, 2012.

[101] Mikolov, T., KarafiÃąt, M., Burget, L., Cernockáżş, J., and Khudanpur, S. Recurrent neural network based language model. In *INTERSPEECH* (2010), 1045–1048.

[102] Mikolov, T., Sutskever, I., Deoras, A., Le, H.-S., Kombrink, S., and Cernocky, J. Subword language modeling with neural networks. *preprint (http://www. fit. vutbr. cz/imikolov/rnnlm/char. pdf)* (2012).

[103] Molina, B. Sega reveals data breach affecting 1.3 million users. *USATODAY.COM* (June 2011).

[104] Monrose, F., and Rubin, A. Authentication via keystroke dynamics. In *CCS '97: Proceedings of the 4th ACM Conference on Computer and Communications Security*, ACM (New York, NY, USA, 1997), 48–56.

[105] Myrvold, W., and Ruskey, F. Ranking and unranking permutations in linear time. *Information Processing Letters 79*, 6 (2001), 281 – 284.

[106] Narayanan, A., and Shmatikov, V. Fast dictionary attacks on passwords using time-space tradeoff. In *CCS '05: Proceedings of the 12th ACM Conference on Computer and Communications Security*, ACM (2005), 364–372.

[107] Neilson, W. A., Ed. *Webster's new international dictionary of the English language.*, 2d ed., unabridged. ed. Springfield, Mass.,, C. & G. Merriam, 1934.

[108] Nettle, S., O'Neil, S., and Lock, P. *PassWindow: A New Solution to Providing Second Factor Authentication*. VEST Corporation, 2009.

[109] Openwall. Openwall wordlists collection. `http://www.openwall.com/wordlists/`.

[110] Paninski, L. Estimation of entropy and mutual information. *Neural Comput. 15*, 6 (2003), 1191–1253.

[111] Paolacci, G., Chandler, J., and Ipeirotis, P. Running experiments on Amazon Mechanical Turk. *Judgment and Decision Making 5*, 5 (2010), 411–419.

[112] Peckham, M. Is valve's steam hack as bad as the sony PlayStation debacle? *Time* (Nov. 2011).

[113] Perlroth, N. Hackers Release More Data From Stratfor. *The New York Times Bits Blog* (Dec. 2011).

[114] Peto, R., and Peto, J. Asymptotically efficient rank invariant test procedures. *Journal of the Royal Statistical Society. Series A (General)* (1972), 185–207.

[115] Petrov, S., Barrett, L., Thibaux, R., and Klein, D. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, ACL-44, Association for Computational Linguistics (Stroudsburg, PA, USA, 2006), 433–440.

[116] Petrov, S., and Klein, D. Improved inference for unlexicalized parsing. In *Proceedings of NAACL HLT* (2007), 404–411.

[117] Pliam, J. O. On the incomparability of entropy and marginal guesswork in brute-force attacks. In *INDOCRYPT '00: The 1st International Conference on Cryptology in India* (2000).

[118] Protalinski, E. Chinese hacker arrested for leaking 6 million logins. *ZDNet* (Mar. 2012).

[119] Raftery, A. E. Bayesian model selection in social research. *Sociological methodology 25* (1995), 111–164.

[120] Ramsbrock, D., Berthier, R., and Cukier, M. Profiling attacker behavior following SSH compromises. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on* (2007), 119–124.

[121] Rapid7. Linkedin passwords lifted. `www.rapid7.com/resources/infographics/linkedIn-passwords-lifted.html`, retrieved September 2012.

[122] Rashid, F. Y. Anonymous breaches Booz Allen Hamilton to reveal 90,000 military passwords. *eWeek* (July 2011).

[123] Riddle, B. L., Miron, M. S., and Semo, J. A. Passwords in use in a university timesharing environment. *Computers and Security 8*, 7 (1989).

[124] Ryan, T. A. Comment on "Protecting the overall rate of type i errors for pairwise comparisons with an omnibus test statistic": or Fisher's two-stage strategy still does not work. *Psychological Bulletin 88*, 2 (1980), 354–355.

[125] Schneier, B. MySpace passwords aren't so dumb. `www.wired.com/politics/security/commentary/securitymatters/2006/12/72300`, December 2012, retrieved September 2012.

[126] Shannon, C. E. A mathematical theory of communication. In *Bell System Technical Journal*, vol. 7 (1948), 379–423.

[127] Shay, R., Kelley, P. G., Komanduri, S., Mazurek, M. L., Ur, B., Vidas, T., Bauer, L., Christin, N., and Cranor, L. F. Correct horse battery staple: exploring the usability of system-assigned passphrases. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, ACM (New York, NY, USA, 2012), 7:1–7:20.

[128] Shay, R., Komanduri, S., Durity, A. L., Huh, P. S., Mazurek, M. L., Segreti, S. M., Ur, B., Bauer, L., Christin, N., and Cranor, L. F. Can long passwords be secure and usable? In *Proceedings of the 2014 Annual ACM Conference on*

Human Factors in Computing Systems, CHI '14, ACM (New York, NY, USA, 2014), 2927–2936.

[129] Shay, R., Komanduri, S., Kelley, P. G., Leon, P. G., Mazurek, M. L., Bauer, L., Christin, N., and Cranor, L. F. Encountering stronger password requirements: User attitudes and behaviors. In *SOUPS '10: Proceedings of the 6th Symposium on Usable Privacy and Security*, ACM (2010).

[130] Simon, N., Friedman, J., Hastie, T., and Tibshirani, R. Regularization paths for cox's proportional hazards model via coordinate descent. *Journal of Statistical Software 39*, 5 (2011), 1–13.

[131] Steinman, G. Security update for all Ubisoft account holders. `http://blog.ubi.com/security-update-for-all-ubisoft-account-holders/`, July 2013.

[132] Sutskever, I., Martens, J., and Hinton, G. E. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)* (2011), 1017–1024.

[133] Toomim, M., Kriplean, T., Pörtner, C., and Landay, J. Utility of human-computer interactions: Toward a science of preference measurement. In *Proceedings of the 2011 annual conference on Human factors in computing systems* (2011), 2275–2284.

[134] Ur, B., Kelley, P. G., Komanduri, S., Lee, J., Maass, M., Mazurek, M. L., Passaro, T., Shay, R., Vidas, T., Bauer, L., Christin, N., and Cranor, L. F. How does your password measure up? the effect of strength meters on password creation. In *Proceedings of the 21st USENIX conference on Security symposium*, USENIX Security '12, USENIX Association (Berkeley, CA, USA, 2012), 5–5.

[135] Ur, B., Segreti, S. M., Bauer, L., Christin, N., Cranor, L. F., Komanduri, S., Kurilova, D., Mazurek, M. L., Melicher, W., and Shay, R. Measuring real-world accuracies and biases in modeling password guessability. In *Proceedings of the 24th USENIX conference on Security symposium*, USENIX Security '15, USENIX Association (Washington, D.C., Aug. 2015), 463–481.

[136] van Oorschot, P. C., Salehi-Abari, A., and Thorpe, J. Purely automated attacks on PassPoints-style graphical passwords. *IEEE Transactions on Information Forensics and Security 5*, 3 (Sept. 2010), 393–405.

[137] van Oorschot, P. C., and Thorpe, J. On predictive models and user-drawn graphical passwords. *ACM Transactions on Information Systems Security 10*, 4 (2008), 1–33.

[138] van Oorschot, P. C., and Thorpe, J. Exploiting predictability in click-based graphical passwords. *Journal of Computer Security 19*, 4 (June 2011), 669–702.

[139] Vance, A. If your password is 123456, just make it hackme. *The New York Times 20* (2010).

[140] Veras, R., Collins, C., and Thorpe, J. On semantic patterns of passwords and their security impact. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, NDSS '14 (2014).

[141] Vijayan, J. Open source WineHQ database breached. *Computerworld* (Oct. 2011).

[142] Wang, K., Thrasher, C., and Hsu, B.-J. P. Web scale NLP: a case study on url word breaking. In *Proceedings of the 20th international conference on World wide web*, WWW '11, ACM (New York, NY, USA, 2011), 357–366.

[143] Weir, M. Probabilistic password cracker - reusable security tools. `http://sites.google.com/site/reusablesec/Home/password-cracking-tools/probablistic_cracker`.

[144] Weir, M. *Using Probabilistic Techniques To Aid In Password Cracking Attacks*. PhD thesis, Florida State University, 2010.

[145] Weir, M., Aggarwal, S., Collins, M., and Stern, H. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *CCS '10: Proceedings of the 17th ACM Conference on Computer and Communications Security*, ACM (2010), 162–175.

[146] Weir, M., Aggarwal, S., Medeiros, B. d., and Glodek, B. Password cracking using probabilistic context-free grammars. In *SP '09: Proceedings of the 2009 IEEE Symposium on Security and Privacy*, IEEE (2009), 391–405.

[147] Wikipedia. Semiparametric model — wikipedia, the free encyclopedia, 2014. [Online; accessed 11-January-2015].

[148] Wikipedia. Prefix sum — wikipedia, the free encyclopedia, 2015. [Online; accessed 24-October-2015].

[149] Wilcoxon, F. Individual comparisons by ranking methods. *Biometrics bulletin* (1945), 80–83.

[150] Wimberly, H., and Liebrock, L. M. Using fingerprint authentication to reduce system security: An empirical study. In *SP '11: Proceedings of the 2011 IEEE*

Symposium on Security and Privacy, IEEE Computer Society (Washington, DC, USA, 2011), 32–46.

[151] Zhang, Y., Monrose, F., and Reiter, M. K. The security of modern password expiration: An algorithmic framework and empirical analysis. In *CCS '10: Proceedings of the 17th ACM Conference on Computer and Communications Security*, ACM (2010), 176–186.

[152] Zviran, M., and Haga, W. J. Password security: an empirical study. *Journal of Management Information Systems 15*, 4 (1999), 161–185.