Chapter 2
The Benefits of Understanding Passwords

Markus Jakobsson, Mayank Dhiman

Abstract

In an effort to assess the strength of passwords, password strength checkers count lower-case and upper-case letters, digits and other characters. However, this does not truly measure how likely a given password is. To determine the likelihood of a password, one must first understand how passwords are generated – this chapter takes a first step in that direction. This is particularly important in a mobile context, where users already are tempted to use short and simple passwords – given how arduous password entry is.

2.1 Why We Need to Understand Passwords

While we do not think that passwords are the best way for people to authenticate to their devices and service providers, it is important to recognize the degree to which passwords are part of infrastructure, which makes them difficult to replace – even if we agree on what to replace them with.

This chapter describes a new method by which we can address two common problems relating to traditional passwords. The first problem is that of approximating the security of a given credential. Traditional password strength checkers plainly demand the presence of certain predicates – such as a combination of uppercase and lowercase; the inclusion of numerals; and that passwords do not match any of the most common passwords (such as "abc123"). This is not necessarily the optimal strategy, as it does not capture common transformations (such as from to common password "password" to the very similar "passw0rd"). Rather than extending the blacklist to all common variants of all common passwords, it is better to understand the underlying structure of passwords, and how people generate them. This allows us to score passwords based on how they were generated – doing this allows us to determine that "p1a2s3s4w5o6r7d" is somehow less secure than "a1d9o8g4." Without an understanding of how passwords are generated, the former password is likely to be believed to be the stronger of the two. (Of course, if mindless exhaustive search is the only path of compromise, the former password *is* the strongest of the two – so security must be seen in context of the most prevalent threat.)

The second problem this chapter addresses is how to identify credential reuse – whether sequentially for one account, or consecutively between two or more accounts. Here, we do not only consider *verbatim* reuse, but also *approximate* reuse – such as “BigTomato” and “bigTOMATO1.” If a person has two accounts with different passwords then loses one of the passwords to a fraudster, then the fraudster has a reasonable chance to get access to the other account as well. This is because the attacker may try all common transformations of the stolen credential, hoping that one of them will work for the second account. As a result, we need to identify and discourage both verbatim and approximate password reuse. While verbatim reuse can be detected without any understanding of the underlying credentials, detection of approximate reuse requires a structural understanding of passwords.

The two techniques described herein are closely related, and are both based on the parsing and decomposition of passwords, using rules matching those people are relying on when they *generate* passwords. Examples of such rules are insertion of one component into another component; concatenation of components; and common transformations of elements of a component.

2.2 People Make Passwords

A good password is hard to guess. Conversely, of course, a bad password is *easy* to guess. But what is it that makes something hard to guess, and how can we tell?

It is easier to tell that something is *easy* to guess than that it is hard to guess. For example, the following potential passwords are easy to guess: *fraternity* (a dictionary word); *$L* (a very short string); *qwertyuiop* (a string with a very predictable pattern); and *LoveLoveMeDo* (famous lyrics). Similarly, one can look at the commonality of passwords – any user who wants to use a password that has already reached the limit has to think of another password. This approach is taken by Schechter, Herley, and Mitzenmacher [81]. We can make a long list of reasons to consider a password to be weak – and this is what typical password strength checkers do – but how can we tell that we have not missed some?

To be able to determine what makes most sense, we need to understand how passwords are constructed. Passwords are constructed by *people*, and people follow guidelines and mental protocols when performing tasks. Therefore, a better understanding of passwords requires a better understanding of people – or at least how people construct passwords.

To gain a better understanding of this, we collected a very large number of actual passwords. We sampled and reviewed these, thinking carefully about how each password was constructed. It is meaningful to think of passwords as strings that are composed of *components*, where components are *dictionary words*, *numbers*, and *other characters*. When producing a password, a typical user composes a password from a small number of such components using one or more *rules*.

The most common rule is *concatenation*, followed by *replacement*, *spelling mistake*, and *insertion*. Here, an example of a concatenation is producing “passbay1” from the three components "pass," "bay," and "1." Use of *L33T*[1] is a common replacement strategy, creating "s3v3nty" from "seventy" by replacement of each "e" with a "3." Misspellings may be intentional or unintentional, resulting in passwords such as "clostrofobic." Finally, insertion produces strings such as "Christi77na," where "77" was inserted into the name "Christina." (This was the least common type of rule among those we surveyed and the hardest to automatically detect in practice, so this rule was not used in the experiment we describe herein.)

The simple insight that *people* choose passwords suggests a new approach to determining the strength of a password: One can determine the components making up the password and the commonality of each such component; one could then consider the mental generation rules used to combine the components and make up the password – along with the commonality of these rules being used. The strength of the password, in some sense, depends directly on the commonality of the password, which in turn depends on the commonality of its components and password generation rules. Similarly, when determining the similarity of two passwords, one can compare the components that make up the two passwords along with the generation rules. It is therefore important to understand the commonality of components and rules.

2.3 Building a Parser

Components and Rules

To build a parser, we need to understand the components and the generation rules, and then "decompile" passwords into the components they were made from. We will therefore review how most passwords are formed, in order to understand how to invert this process.

Examples

- **Concatenating components.** The password "mylove123" consists of three components: "my," "love," and "123" combined with two occurrences of the concatenation rule. By determining the observed frequencies of the three components and the concatenation rule, it is possible to assess the "likelihood" of the password.
- **Order matters.** The password "my123love" has the same components and uses the same rules – except it does not attach the numerals at the end. A simple analysis of passwords shows that most passwords containing numerals have them at the end. And since the goal of parsing is to assess the likelihood of a given password – its believed strength – we see that order matters. Put a different way,

[1] L33T is pronounced "leet," and is a relatively common transcription of words in which letters are replaced by other characters with some resemblance to the replaced letter.

the “value” of the component “123” is greater when it is not at the end of the password.
- **Insertion.** Next, the password “mylo123ve” is even stronger, given that it does not only use concatenation but also insertion: the user has inserted the component “123” inside the component “love,” which, in turn, was concatenated to the component “my.”
- **L33T and multiple parsing possibilities.** The password “myl0v3” has two components, “my” and “l0v3,” where the second component is L33T for “love.” Here however, “l0v3” can either be seen as a dictionary word whose frequency is recorded and used to score passwords in which it occurs, or it can be seen as the result of a L33T-rule applied to a dictionary word. While the former approach is more straightforward, it does not benefit from knowledge of the observed frequencies of the underlying words. The latter approach, on the other hand, may result in errors where the L33T term is more common – relatively speaking – than the associated word. One approach to address this is to parse the password both ways and use the more conservative assessment.
- **Maximizing coverage.** Finally, the password “thinput” shows that there could be multiple paths. This could either be the result of concatenating the two character components “t” and “h” with the component “input,” or it could be the result of concatenating “thin” and “put.” Like in the previous example, it is possible to produce multiple parsing results and then select the most conservative. In the parser we describe, though, a greedy word-based approach was taken, where passwords are considered combinations of the words that “cover” the biggest portion of the strings. This is a result of the observation that most passwords are based on words, which surely is due to the fact that people relate better to words with meaning than to strings of random characters.

What the Parser Does

The parser takes a password as input and outputs the various components and rules which have been used to construct that password – or *appear* to have been used, to be specific. As we have mentioned, components include words, numbers, individual characters, and special symbols. Some components may be overlapping, e.g., a word is made up of number of characters. In such cases, those components are parsed first which have longer length. This minimizes the number of components that are used to “cover” the credential.

The parser has built-in rules to identify three major password creation rules: *concatenation*, *L33T*, and *misspelling.*

In order to parse an input, the parser uses an *input dictionary*. It is worth noting that language has strong influence on the creation of passwords and that words usually form the core around which various other operations are applied to make the password more complex. For simplicity, only English dictionaries are used herein.

The *input dictionary* contains words from different sources, such as a standard English dictionary, people's first and last names, geographical locations, technical terminology, and so forth. A vanilla English dictionary is not sufficient because many passwords contain words and phrases from the latter categories. To achieve a better and more comprehensive dictionary, we combined specialized dictionaries based on such topics. The final dictionary obtained upon merging these specialized dictionaries contained just below 670,000 words.

The parser contains algorithms capable of identifying password creation rules. The parser uses the input dictionary and the specified algorithms to identify the components and rules for the input password. The detection of components is directly dependent upon the choice of input dictionary.

Note that in many cases the passwords can be broken into components in more than one way. In such cases, the parser selects the components containing words with the *maximum coverage*. Maximum coverage means that the total length of the combined components, which are words, is maximized. For example, "thinput" will get parsed as "thin" and "put" rather than "t," "h," and "input." When two paths produce the same coverage, the path with the greatest probability of occurrence (as judged by the frequency of use of the rules and components) is chosen. Practically speaking, this typically corresponds to the path with the smallest number of components. Therefore, "password123" will be parsed as "password" and "123," rather than "pass," "word," and "123." The details of how the parser operates are given the next subsection.

Parsing Details

The three main subroutines which are required for parsing components are *generate-all-substrings*, *generate-max-substrings* and *generate-leftover-components*, which we will describe in detail next.

Consider a potential password "hello1." A call to *generate-all-substrings* will generate all possible substrings of a given string. Hence, a call to *generate-all-substrings* using this input generates "h," "e," "l," "o," "1," "he," "el," "ll," ..., "hello," "ello1," "hello1."

The *generate-max-substrings* subroutine takes in a password as an input and generates a set of substrings, which are words and can be found in the input dictionary. Only that set of substrings is returned which provides the *maximum coverage* of the input string. For example, using *generate-all-substrings* as a subroutine, and an input "hello2(world!35b," the subroutine *generate-max-substrings* returns a list containing "hello" and "world."

The third subroutine *generate-leftover-components* requires two arguments: the password and the list of substrings generated by *generate-max-substrings* using the password as an input. This subroutine will return the remaining components – i.e., consecutive numbers, characters and special symbols – after cutting the substrings generated by *generate-max-substrings* from the password. Thus, calling *generate-*

leftover-components with the inputs of "hello2(world!35b" and ["hello," "world"] returns "2," "(,""!," "35," and "b." Notice that the string is broken up into conceptually consistent pieces – e.g., "!35b" is split into "!," "35," and "b."

Regular Components and Concatenation

In order to identify the components, the parser first checks if the password can be found in the input dictionary. In that case, the password contains only one component, the password itself. For example, "monkey" is one of the most commonly used passwords, which can be found in the dictionary. In the next step, the parser checks if the password contains numbers only. For example, keyboard patterns like "12345" and dates of birth are quite common passwords. The first two steps are performed in this order due to efficiency reasons. This is because an average of 25% of all passwords in most datasets are either dictionary words or complete numbers.

In order to detect concatenation, the parser uses the *generate-max-substrings* subroutine described above, which generates the set of substrings that provide *maximum coverage*. These results are combined with the results of *generate-leftover-components* to generate all the components of the password.

Identifying L33T

Since, L33T is not a proper language and many different dialects of L33T occur on the Internet, there is no perfect algorithm to convert between English and L33T. Since the rules for such conversion vary, one can use an exhaustive approach. One can construct a table containing all mappings from a token to be replaced (number or special symbol) to a list of all common replacements for that token, as seen in various L33T to English translators. For example, the character "0" is substituted for "o" and the character "|" can be substituted for both "i" and "l." Hence, in the mapping table there will be an entry for "@" to be mapped to "a" and "|" to be mapped to both "i" and "l."

Identification of L33T works in combination with the *generate-max-substrings* subroutine. The *generate-max-substrings* subroutine tries to replace the tokens with all possible characters found in the mapping of token in the L33T to English mapping table. All possible combinations are tried. For each combination tried, the parser tries to find the maximum covering strings. If the new substrings generated covers a larger length than the previously calculated substrings, then that character is replaced by the new character from the mapping. Consider the password "||oveyOu." Each token which is not a letter is considered for replacement – in this example, there are three such characters. For each combination of "reasonable" substitutions, *generate-max-substrings* is run. The substitution with the greatest coverage is selected and output; in our example, "||oveyOu" would be converted to "iloveyou," which then would broken into the three words it consists of.

Identifying Spelling Mistakes

Identification of spelling mistakes is more complex. Initially, the *generate-max-substrings* subroutine is called and the output of it is then used as input for the *generate-leftover-components* subroutine in order to examine leftover components. For example, consider the password “heilloworld.” A call to *generate-max-substrings* will generate “world” which along with the initial password is used as an input to *generate-leftover-components* which will generate “heillo.” This is the leftover string and potentially contains a spelling mistake. Before starting to parse the passwords, a training module is used to train for detection of spelling mistakes using an input dictionary. There is another component of this training module, which tries to detect and correct the spelling of each component, if possible. In case a spelling mistake is found, then a new string is generated in which the spelling mistake has been rectified, and the subroutine *generate-max-substrings* is called for a second time using this new string. This is because in certain cases, the incorrectly spelled word may be used to create longer words. For example, “jidgement” will initially generate the component “gem,” but after the spelling mistake has been rectified, calling *generate-max-substrings* will generate “judgement.”

2.4 Building a Model

The parser we have described can be used to process large sets of passwords, producing the associated components and rules used to produce the passwords in the sets. By recording the frequencies of each of the components observed – and the rules that were used – one can produce a stochastic model for how passwords are generated. This model can then be used to score passwords. In the following, we will describe how to build the model.

Approach

In order to build a model of passwords, one needs a large collection of passwords and an input dictionary. One can use any set of passwords to train the system. The larger the dataset is, the more accurate the resulting frequency database will be. If one small dataset is considered higher quality – i.e., more accurate – than a larger dataset, then one can perform “weighted” training using both. In the example we describe, however, we only used one dataset for training: The *RockYou dataset*. The RockYou dataset contains 32 million passwords, leaked in 2007. During training, we parse passwords from the RockYou dataset and generate various rules and components. In the process, we populate our input dictionary with the frequencies of occurrences of each rule and component.

After this training phase, we obtain three *trained dictionaries* containing the component/rule and the corresponding frequency of occurrence. These include a dictionary of words, a dictionary of numerals, and a dictionary of characters and symbols. Upon breaking down a password into the components and rules, the score calculator makes a search count for the components and rules in the *trained dictionaries*. Using these values, a score corresponding to the input password is calculated using the scoring algorithm.

Characteristics of Password Collections

After deriving a model by training the system with a large number of passwords, this model can be used to assess the strength of passwords. We test this by computing password scores from five datasets, each one of which corresponds to a collection of passwords. Each dataset has different characteristics, as will be shown, which influences the average strength of the passwords and hence, the password scores. We first describe three major characteristics of all datasets.

The first obvious characteristic is the *type of resource being protected*, i.e., what would be lost if a password is stolen. We consider only the losses as perceived by the user who produces the password, and not those potentially suffered by other users as a result of theft or the losses of the organization associated with the password. The rationale is this: users are influenced by the risk they associate with theft when they select their password. For financial service providers, money would be lost. For social networking sites, the user would lose face or be inconvenienced. For a porn site, the loss may be limited to the access to the site. In addition to this, some private information, such as user name and address, may be at stake for all of these types of sites.

A second characteristic is the *demographics of users*. While it is not known how demographics affect password strength, it appears likely that they do.

Finally, the *collection method* could introduce a bias in the dataset. For example, if a dataset is obtained by malware attacks, the dataset may have a greater percentage of passwords from people who are not security conscious than if it was obtained by site corruption. Similarly, a dataset associated with phishing is likely to have a greater percentage of passwords from people who are gullible than other datasets would.

In the following, we will describe results associated with five datasets. The first dataset, the *Rootkit dataset* contains 64498 passwords. It was obtained by a compromise of rootkit.com. This website has forums about discussions of advanced topics in computer security. The demographics introduce a bias as relative to an average user, as most people registering on this website are more aware about security issues than typical users. Second, the *Paypal dataset* contains 19053 passwords. Most of the users behind the passwords are adults. The method of collection is phishing, which introduces a bias towards people who are more gullible. Third, the *Justin Bieber dataset* contains 5091 passwords and was obtained by a compromise of a fan website. Hence, there is no bias due to the collection method. However, the demographics introduce a bias as most of the users are teenagers. The fourth dataset is the *Sony dataset*, which contains 17785 passwords, and was obtained by a compromise of Sony Pictures Europe. As a result, there was no bias among passwords due to the collection method. And finally, the *Porn dataset* contains 8089 passwords, and was obtained by a compromise of a pornographic website. As a result, there is no bias due to the collection method. However, there is a visible bias associated with the type of resource being protected.

Training

Once we have decided with the input dictionary and the training dataset, which in our case is the RockYou dataset, we proceed to the training phase. Since, the dataset is huge, containing a total of 32 million passwords, it is first arranged as a tuple *(password, count)*. In this manner, passwords can be parsed quicker. Then the parser parses the dataset and creates three *trained dictionaries*. As mentioned earlier, three different trained dictionaries – a dictionary of words, a dictionary of characters and special symbols and a dictionary of numerals – are obtained after training on the dataset. The main reason to create three different populated dictionaries than one is that at the time of calculating the scores, the components may have a little overlap. For example, while parsing a password like "Spassword," it is better to first look for words rather than individual characters and hence, the word "password" should be parsed first followed by individual characters in order to obtain *maximum coverage*. Similarly, in the case of a password like "passw0rd," words should be parsed before numerals. Then it would be possible to detect L33T and hence, the word "password" with larger coverage could be obtained as compared to the word "pass" with smaller coverage. Thus, parsing will be more efficient if three different dictionaries are maintained against one.

2.5 Scoring Passwords

Scoring of passwords is straightforward once we have a parser and a list of frequencies of components and rules. We simply parse an input password and determine the frequency of each component and rule. Multiplying all of these frequencies with each other gives us a measure of the likelihood of the password.

Results

In this subsection, we analyze the usage of various rules and components in datasets described above. Using these analysis, we can better analyze the perceived security risks associated with these websites. This may lead to a better understanding of user behavior regarding creation of passwords. For example, it maybe useful to answer questions like how people's password creation policies change when the security risks involved are changed, and hoow various factors like type of resource being protected and demographics may influence password creation policies. These analysis also provide clues to a better password strength checker. We begin by comparing the usage of various rules and components in the datasets and then move onto analysis of passwords in individual datasets.

The Frequencies of Components: Words

Word components form the core of a large percentage of passwords, as can be seen in figure 2.1. The average number of word components per password is directly related to the average strength of the password. The Justin Bieber dataset has the highest average number of word components per password, whereas the Porn dataset has the lowest. It is not surprising that the Porn dataset is associated with a low average password security – after all, the loss of a password does not cause any great damage to a user, except if the user employs the same password elsewhere. However, it is a bit surprising that the Justin Bieber dataset exhibits a greater password security than the Paypal dataset – after all, the protected resources for the former are simply a user profile on a fan site. The reason may be that the Justin Bieber passwords were corrupted at the site while the Paypal passwords were stolen from gullible users, who are likely to use lower quality passwords than more security-conscious users are. It may also be due to demographic differences. At the same time, it is important to recognize that average quality is not a measure of how vulnerable the most exposed users are. As a case in point, it can be seen that a small number of words in the Justin Bieber dataset were used to construct a much larger number of passwords than any other dataset – see figure 2.2 for an illustration of this. This has a negative contribution to the average strength of passwords and counters the higher usage of word components per password in the Justin Bieber dataset. This is quite evident from table 2.1, where the average product of frequencies of all components per password is not the highest in the Justin Bieber dataset as the usage of word components is highly skewed. This forms maybe the best argument for why one should parse passwords. By doing so, it is possible to block the use of the most vulnerable passwords.

Table 2.1 The *average security* i.e., the average product of component frequencies for the passwords. The values are as expected with the Porn dataset taking the lowest value. The only exception is the Sony dataset having the largest average product.

Dataset	Security
JB	17.52
Rootkit	15.09
Paypal	15.77
Sony	18.67
Porn	13.79

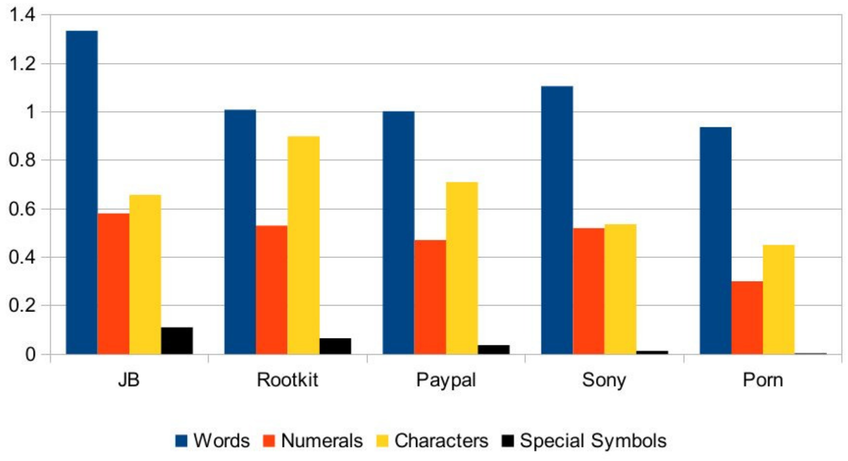

Fig. 2.1 The figure shows the average number of components per password in the different datasets.

The frequencies of components: Numerals

Common wisdom – expressed in the rules of password strength checkers – hold that it is good to include numerals in passwords. At the same time, it is clear that using numerals only make for weaker passwords. This is illustrated by figure 2.4, which visualizes a particular weakness of passwords in the Rootkit dataset: approximately 15% of all passwords in Rootkit dataset are numeric. If proposed passwords were decomposed as described herein, such weaknesses could easily be avoided. The use of numerals as components in passwords in general is illustrated in figure 2.1.

The frequencies of components: "Other"

The remaining components include leftover characters and special symbols obtained after "cutting" words and numerals from passwords. Even though usage of special symbols in passwords increases the effective size of the character set, special symbols are unpopular among users – see figure 2.1.

Even in the small percentage of special symbols which are used, the usage is highly skewed. Out of the 32 possible special symbols on standard ASCII keyboards, only a few of them have high frequencies. These special symbols are usually those which are used in rules like L33T – e.g., "@" and "&." Most of the remaining special symbols are rarely used. In fact, we found some special symbols which were almost never used in any of the datasets, such as "?" and "}." This is not a statistic anomaly given that the datasets contain over 100,000 passwords. It's a reminder that humans do not select passwords randomly, but rather, based on mnemonics and rules used by many others.

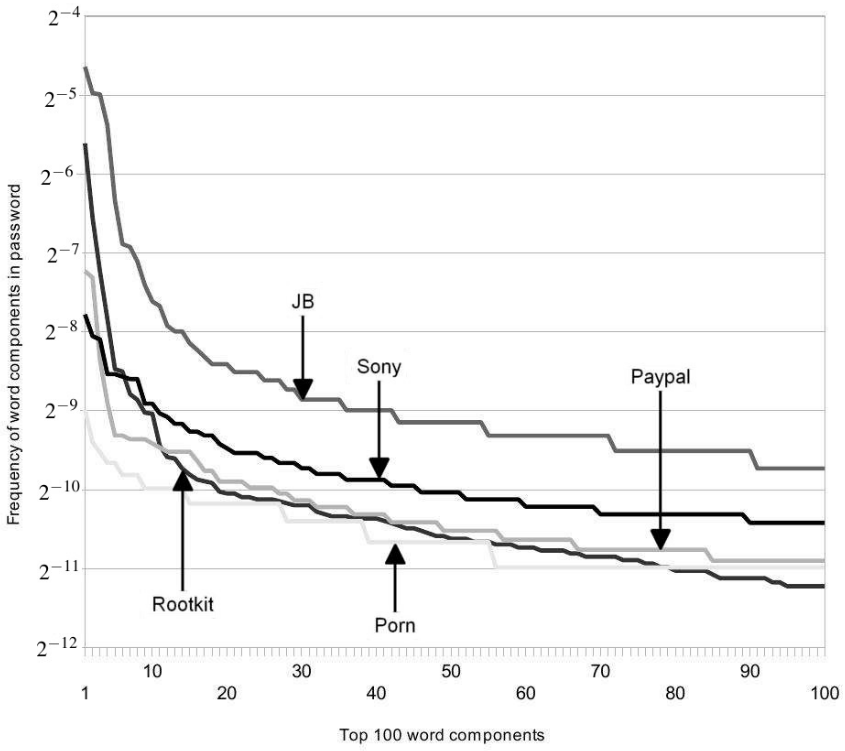

Fig. 2.2 The figure shows the frequencies of the top 100 most common words in each dataset. Even though passwords from the Justin Bieber dataset usually contain a larger number of components per password, it can be clearly seen that a small number of words are reused to a greater extent within that dataset.

The frequencies of rules: Concatenation

For a given password, the number of concatenation operations is one less than the total number of components. It can be clearly seen from figure 2.5 that concatenation is prevalent in all datasets. In fact, concatenation is the most used basic operation in all of the datasets. In the Justin Bieber dataset, it is as high as 1.67 concatenations, i.e., an average of 1.67 components per password. The higher usage of concatenation associated with higher security sites also indicates how people increase password security.

The Frequencies of Rules: L33T

Usage of L33T requires users to remember a new pattern replacing an old pattern which may require some effort, but can dramatically increase the strength of the

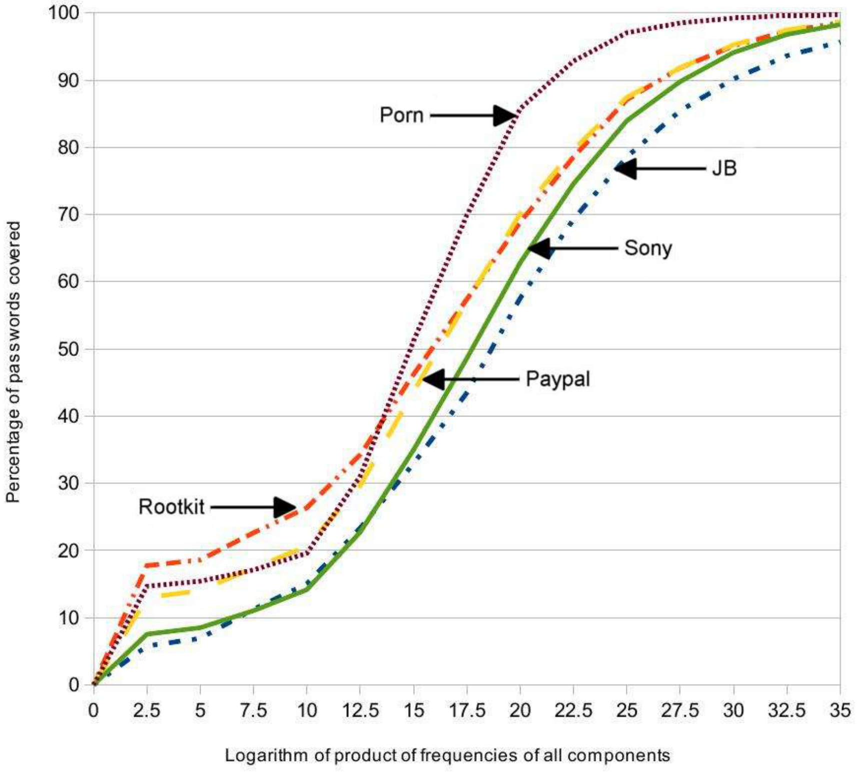

Fig. 2.3 The figure plots the cumulative distribution of the product of frequencies of all components of the passwords. This product is an estimate of the bit strength of the passwords. The Porn dataset stands out as having weaker passwords than the other datasets. By decomposing passwords into components and applying rules for minimum strength, it is possible to avoid weak passwords.

password. The average number of L33T operations per password is shown in figure 2.5. The highest value is for the Rootkits dataset, which supports our belief that there are demographic differences between password datasets, and that in particular, "geeks" are more frequent users of L33T than others.

Considering the scoring, if we were to assign a greater weight to using L33T than concatenation, we could "reward" users who use uncommon rules.

The Frequencies of Rules: Spelling Mistakes

The usage of spelling mistakes is quite interesting. To begin with, it is not possible for us to determine whether spelling mistakes are intentional or not, whereas the use of other rules are evidently intentional. Unintentional spelling mistakes, in the context of password strength, is therefore an excellent example of a case where ig-

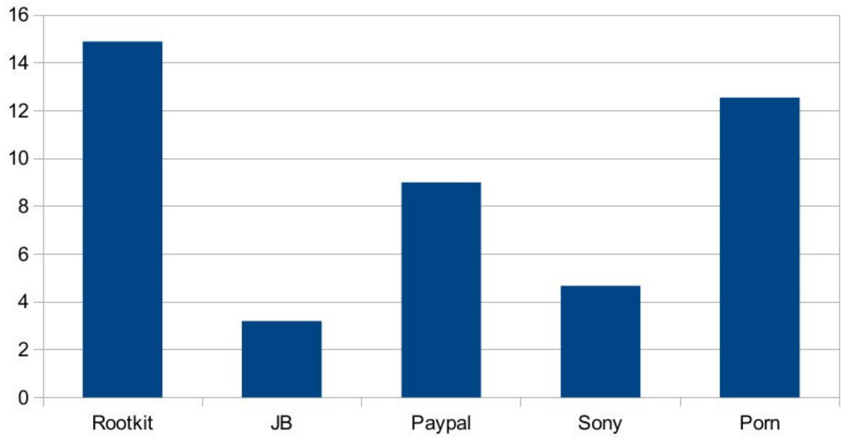

Fig. 2.4 The figure shows the percentage of passwords that are complete numerals. While this contradicts the current password rules of Paypal passwords, where complete numerals are not permitted – many of the captured passwords predate this rule. By applying password strength assessments like the one we propose, not only when the password is first created, but also during login, weak and non-compliant passwords can be tagged and users be asked to update passwords.

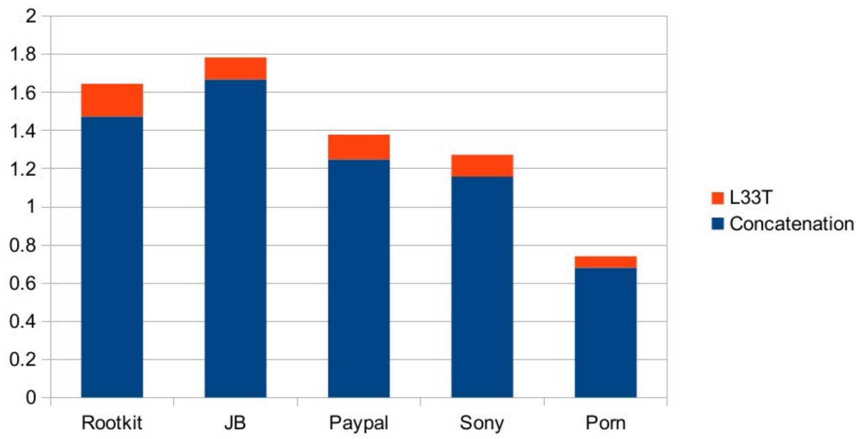

Fig. 2.5 The average use of concatenation and L33T rules per password for different datasets. L33T (pronounced “leet”) is the replacement of characters with similar-looking characters – for example, making “@ppl3” from “apple.”

norance is bliss. Figure 2.6 shows the prevalence of spelling mistakes in the different datasets with much greater frequency in the Porn dataset.

While the different datasets exhibit different rates of spelling errors, they are all relatively low in comparison to other rules. The low frequency of the use of spelling mistakes emphasizes the potential value of this rule to password security.

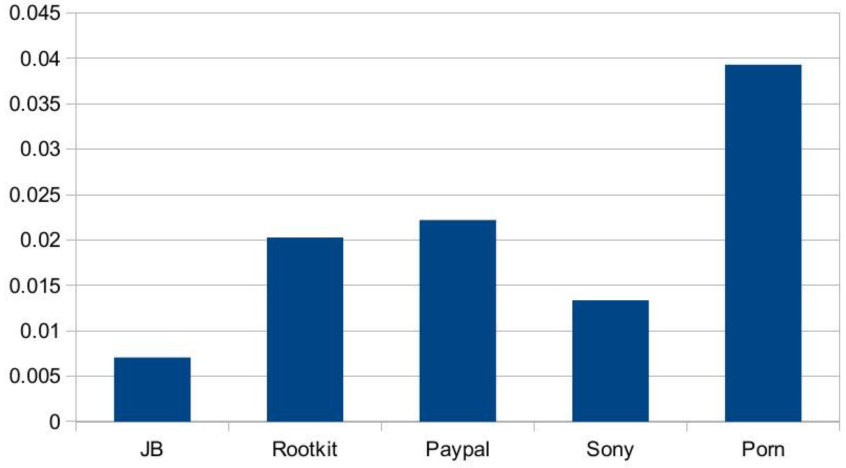

Fig. 2.6 The average number of spelling mistakes per password for the different datasets. The use of this rule is dramatically less common than both L33T and concatenation.

Score Calculation.

The score calculator we describe here uses only the frequencies of rules and components, not the location or order of them. The score calculator utilizes the three different trained dictionaries of words, characters and special symbols, and numerals. All of these are obtained by training on the RockYou dataset.

The frequencies are measured by dividing the count of occurrences by the total count. The score calculator utilizes the frequencies of rule occurrences in that dataset. These are obtained by analysis of passwords of each particular dataset. The score of a password is simply calculated by multiplying the frequencies of all the rules and components occurring in the password. Since all frequencies are between 0 and 1, the score will also be a small value in range of 0 and 1. The second logarithm of this product is an assessment of the bit strength of a password.

Comparison of Scores.

Figure 2.7 shows the distribution of password scores for the passwords of the different datasets. Note the close resemblance to a bell curve. Password crackers target the most frequent credentials, corresponding to the passwords with the lowest scores. The clearly insecure region from figure 2.7 is magnified in figure 2.8. The cumulative distribution of password scores is shown in figure 2.9.

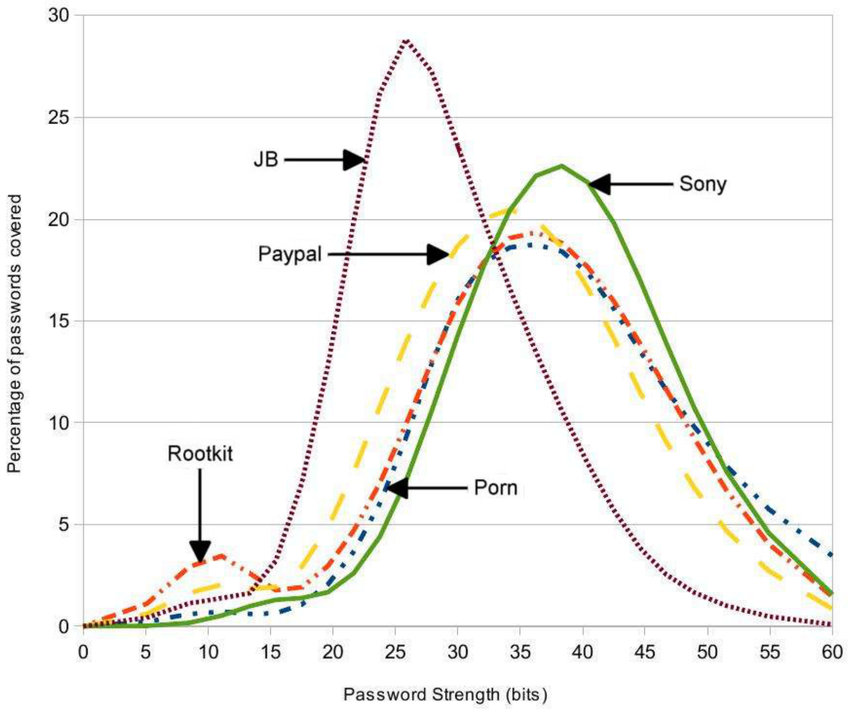

Fig. 2.7 The figure shows the distribution of password strengths as assessed by multiplying the frequencies of rules and components used. The second logarithm of these values correspond to the bit strength of the passwords. It is evident that passwords from the Porn dataset are weaker than the passwords of the other datasets.

2.6 Identifying Similarity

One of the big vulnerabilities associated with typical password use is the common reuse of passwords across multiple sites. Users have accounts on various websites and each website requires them to remember a username, password combination for authentication. As the number of accounts increases, it becomes quite difficult for a user to remember all her passwords. Hence, people reuse passwords. Password reuse is usually not a good idea as different websites have different security threats related to them. A phisher who compromises the password of an email account has a very good chance of being able to access the financial accounts of the same user, provided he can determine the user name used for those. (This may be evident from scanning the mailbox from emails from financial service providers.) At the same time, the understanding of the degree of password reuse is very poor to date. We offer a glimpse at the magnitude of this problem in this section.

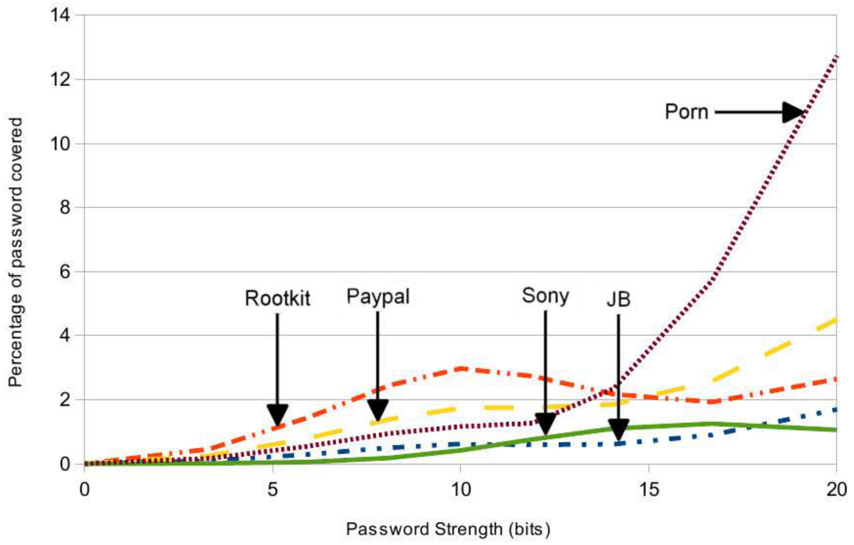

Fig. 2.8 This figure shows a portion of the graph in figure 2.7. This portion of the distribution corresponds to unnecessarily weak passwords. By decomposing passwords and scoring them in the manner we describe, such passwords can be avoided.

An average user has 6.5 passwords, each of which is shared across 3.9 different websites [28]. A user with 25 accounts therefore only remembers the 6 or 7 basic passwords he or she uses, and what password is used where.

Small modifications to existent passwords are also used to generate new passwords. One of the ways in which users tackle the problem of remembering a large number of passwords for their different accounts is by crafting interrelated or similar passwords. A base password is used and then reused with slight modifications to generate passwords for other websites – and to comply with differing password rules. Therefore, even though different accounts owned by one user may not share the same password, the passwords can be quite similar to each other. Fraudsters know this and are likely to try common variations of passwords they steal. For example, a user may choose a base password, ‘password’ and generate new passwords like ‘password123’ (numeral concatenation), ‘passwordabc’ (concatenation), ‘Password’ (capitalization), ‘passw0rd’ (L33T), and so forth. Thus, users generally introduce new components or rules, or modify existing components in order to generate new passwords. (Sadly, these are real examples of passwords and not all too uncommon.)

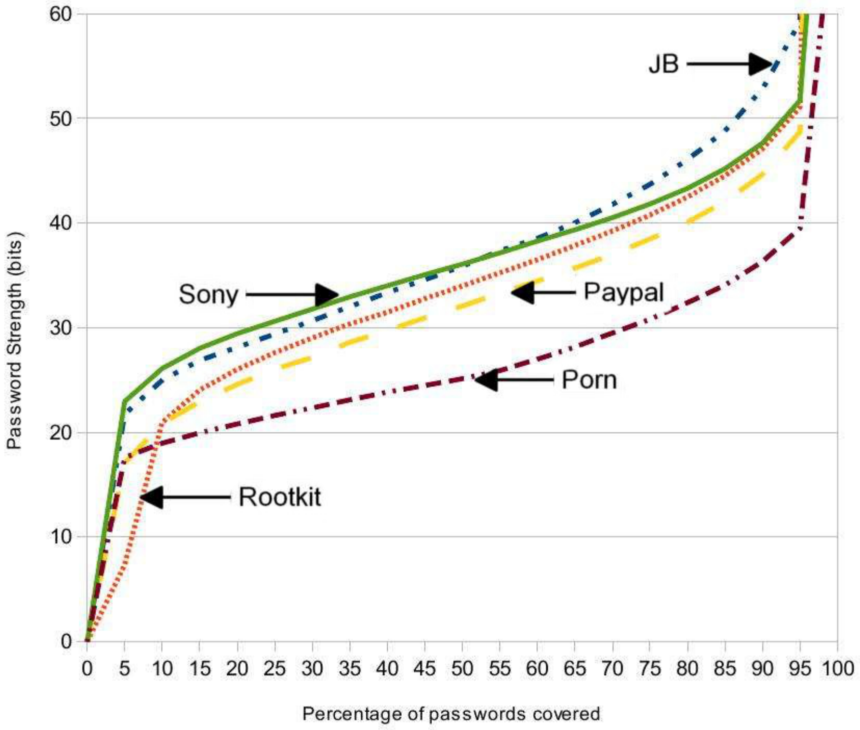

Fig. 2.9 The figure shows the cumulative distribution of password strengths as assessed by multiplying the frequencies of rules and components used. The second logarithm of these values correspond to the bit strength of the passwords.

Dataset

In our analysis, we identify verbatim as well as approximate password reuse, using a dataset obtained by Paypal from a security vendor. The passwords were stolen by a malware, most likely using a key logger that was triggered by the establishment of SSL, or interaction with a password field. These stolen passwords were stored in a dropbox, as commonly done by fraudsters, which in turn was raided by the security vendor to determine what accounts to flag. The dataset contains over 11,000 passwords stolen from 3,550 users, corresponding to an average of 3.174 passwords per user.

Similarity of passwords.

Table 2.2 shows the extent of password reuse. For users for whom only two passwords were compromised, for example, these two passwords were identical for 17% of the users. Similarly, for users for whom three passwords were compromised, at

Table 2.2 The table shows the probability of identifying password reuse between at least two passwords, provided a varying number of observed passwords of a user.

Number of passwords available	Two Passwords are same
Two passwords	17%
Three passwords	36%
Four passwords	71%
Five passwords	85%
Six or more passwords	100%

least two passwords were identical in 36% of the cases. When we get to users for whom six or more passwords were compromised, we see a 100It is evident that password reuse is rampant!

Table 2.3 The table shows the average *Levenshtein distance* of password pairs. As the Levenshtein distance increases, the percentage of passwords having Levenshtein distance decreases. This means that most people who use similar but not identical passwords at different sites make relatively small changes when producing new passwords.

Levenshtein distance	Percentage of Passwords
L 1	5.81%
L 2	5.14%
L 3	2.78%
L 4	2.72%
L 5	2.24%

Table 2.4 Variation of Hamming distance with number of passwords. Unlike Levenshtein distance, the percentage of passwords decreases until HM 3, then increases. We believe that this is directly influenced by the similar structure of many English words. This also suggests that Levenshtein distance may be a better measure to calculate password similarity than Hamming distance.

Hamming Distance	Percentage of Passwords
HM 1	3.93%
HM 2	1.51%
HM 3	1.27%
HM 4	2.84%
HM 5	5.99%

However, this table is not a true reflection of extent of password reuse as it doesn't include those passwords which are similar but not identical. We determined that passwords not identical had high levels of similarity. This was done by computing the *Hamming distance* and *Levenshtein distance* of pairs of passwords of the same user. The Levenshtein distance between two strings is a basic edit distance function that corresponds to the minimum edit distance that transforms a first string into a second string, where the number of deletions, insertions, and substitutions are counted. The Hamming distance is defined as the number of characters which differ between two strings, i.e. the number of characters which need to be changed to turn one string into the other. We refer to tables 2.3 and 2.4 for our observations.

If the Levenshtein distance is less than or equal to four, there is a strong similarity of passwords. This was manually verified as well: the passwords with low Levenshtein distance were confirmed to be derivations of each other. This manual analysis identified that common techniques used to derive passwords from each other were capitalization, L33T, and numeral concatenation.