CrystalPass: A Pattern-Based Password Generator

Joseph Bursey
University of California, Irvine
jbursey@uci.edu

Abstract

Passwords are simultaneously the most ubiquitous feature and problem with online user authentication. Services require that users make new passwords for each account, but the problem of password generation falls to the user. In many cases, the user is stuck re-using passwords or using weak ones making themselves vulnerable to password cracking attacks. To help mitigate this issue for users, we present CrystalPass, a novel pattern-based password generator. The user can create and tweak a password to their liking and with the help of accurate entropy feedback, and CrystalPass will generate the password for them. Even better, the patterns can be re-used to make it easier for the user. In this work we will demsonstrate the security of such passwords as well as carry out a user study to discover users' preference for this new kind of password generator.

1 Introduction

Passwords are the most ubiquitous form of user authentication, yet simultaneously the most problematic. In order to maintain the security of their accounts, users are required to create strong passwords and maintain them in a secure manner. However, this proves to be difficult for many users who may not have the time, may misunderstand what makes a strong password, or may not even understand that their passwords are insecure. Even though users should maintain a different password for each account they control, many are left reusing passwords or are forced to create weaker ones they can remember. Furthermore, even users with the best of intentions may create passwords with unseen biases or patterns that make them easier for attackers to guess. On top of this, many companies will enforce that their users or employees change their passwords from time-to-time. Users may not want to put in the necessary effort that makes these practices useful, so they find ways to only slightly change their current passwords. This leads to passwords that are easily guessed by attackers who may know their victim's practices.

In this project, we aim to help mitigate the issue of password generation through the use of patterns that can be used and re-used to create new passwords on the spot. Using patterns will have a distinct advantage in that a user can learn to use a specific pattern shared among their passwords while their individual passwords remain unique and secure. To accomplish this, we present CrystalPass, a pattern-based password generator. It takes a pattern in the form of a regular expression made up of words, digits, punctuation, and letters, and generates a password which follows that pattern. Since the password is generated by CrystalPass in this manner, it has several advantages over other strategies. First, the passwords are more likely to be memorable to the user as they only need to use a single pattern. Second, there are no human-created biases or patterns in the generated passwords, forcing attackers to guess each password with equal probability. Third, we will be able to give clear and accurate feedback on the strength of the password before it is even generated (*i.e.,* based on the pattern). Since CrystalPass controls password generation, this feedback can be proven accurate against even the strongest attacker.

We implement this idea into a usable and distributable Python application, making it very easy for users to install and begin using[1]. We then perform a user study asking users to use our CrystalPass application to generate strong passwords. Lastly, we present the findings of our users study which cover the users' practices, the usability of CrystalPass, and their preference to using CrystalPass.

In this work, we make the following contributions:

- We build CrystalPass, a pattern-based password generator, which will create passwords that are both proven strong and usable.
- We implement an entropy-based feedback meter into CrystalPass which is able to give accurate password strength feedback based on the provided pattern.

[1]Our project is open-sourced at https://github.com/jtbursey/crystalpass/

- We prove the security of the generated passwords even in the face of an attacker who is knowledgeable of the user's pattern, and a user who has re-used the pattern.
- We perform a brief user study on the usability of our tool and the preference of users to use pattern-based passwords.

2 Background

Passwords are used for almost every instance of authentication in the world today. Even when services use another method as their primary form of authentication, passwords are usually the ubiquitous fallback. However, for all their ease of use from the service provider's perspective, users have a much harder time getting the full security out of passwords.

Consider user A. They have just gotten a new job and now their company wants them to create a password for their work account. A has already had a long day getting settled in their new position, and aren't terribly motivated to create a strong password. So, they simply type in what comes to mind first: `worktime123`. Sadly, this is not an uncommon occurrence as this password has appeared in over 16 password breaches according to haveibeenpwned.com [1]. Now that A associates this with their work account, they may also re-use it when another work related service requires a password. This is a very common case for users creating passwords. Unless they are greatly invested in the security of their account, they will likely choose a password that is simple or re-used from another service.

In order to try to prevent users from creating weak passwords, websites will often employ password requirements to ensure users will make brute-force resistant passwords. For instance, a website might require a user to use at least one capital letter, one number, and one punctuation. However, in this case our user A may just use the password `workTime123&`, which is not a very good password when considering stronger attacks such as dictionary attacks.

2.1 Password Cracking

As weaker passwords are rampant, it has become relatively easy for attackers to try to guess user passwords in attacks known as password cracking. Password cracking usually comes in two variants.

2.1.1 Online Attacks

In this attack, the attacker actively tries to login to the user's account by querying their guesses to the service. In a primitive attack, this ends up being the weaker of the two types of attacks. Typically, services will only let a user incorrectly guess their passwords a set number of times, thus limiting the number of guesses an attacker has. Alternatively, the attacker may notice a flaw in the service and choose to exploit that. In this case, they are not really attacking the password itself, but the service.

2.1.2 Offline Attacks

In this attack, the attacker typically has access to the password database already leaked from the service. This is typically considered to be the stronger attack against passwords since the only thing standing between the attacker and the passwords is raw compute power. Thanks to the rise of cryptocurrencies where users compete to compute hashes, it is not difficult for an attacker to attain 1 TH/s (terahash per second) or one trillion hashes per second. Since hashes are the bottleneck in this type of algorithm, an attacker could easily guess 1 trillion hashes per second.

There are two common methods that services will use to mitigate offline attacks. The first is the use of hashes. Most services will not store user passwords in plaintext, but hash them. A hash is a deterministic, one-way mathematical function that takes an input and gives as output a string of seemingly random numbers. Importantly, the input cannot be determined for any given output. Then, when a user attempts to login to their account, the given password is hashed and compared to the hash in the database. If they match, the user is authenticated. Using hashes greatly slows down attackers from accessing passwords as they will have to make guesses, hash those guesses, and then check if they exist in the database. Additionally, services may salt their hashes. A salt is a random number generated for each stored password. When the password is hashed, the salt is concatenated to it and passed as input to the hash. The purpose of the salt is not to make the attacker guess the salt as well (in fact the salt is stored as plaintext), but to slow down the attacker. Now instead of making one guess, hashing it, and checking it against every hash in the database, the attacker must hash with the salt, and can only check against a single stored hash before getting a new salt and trying again for the next entry. This slows down an attacker by a factor of the number of users in the database. In short, your password is a potato that the website stores as hash browns. And it's even better with a little salt!

2.2 Entropy

Entropy is a thermodynamic concept which measures the disorder of a system. In the case of passwords, we use it to represent the complexity of the password, which is then correlated to the number of guesses required to crack it. Entropy in general is described as [2]:

$$E = \log_2(C) \text{ bits} \tag{1}$$

$$C = |S|^L \tag{2}$$

In equation 1, entropy (E) is a log scale of the number of possible passwords (or possible combinations) C, measured

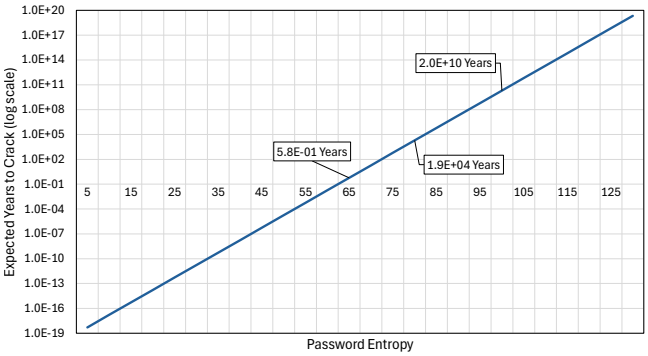

Figure 1: The log-scale time-to-crack for passwords of various entropies.

in bits. The total number of combinations for a password is typically measured by the sets of symbols that are chosen from. These sets are often uppercase and lowercase letters, digits, and punctuation. If at least one symbol from a set is used, the entire set is considered for entropy calculation. In equation 2 above, S is the set of all symbols used in the password, and L is the length. This is how most entropy feedback mechanisms work, and also why they can be so inaccurate. They may not account for words or patterns in the password, and give an overestimated strength as a result. This reveals an important factor in password entropy: the entropy of a password must be based on a specific attacker, and should reasonably be based on the strongest attacker. Otherwise, the entropy will be inaccurate.

Entropy is then directly related to the time it takes for an attacker to crack a single password as shown in Figure 1. Entropy is based on the total number of possible passwords, but this is also the number of passwords an attacker would need to guess in order to crack the password. It is expected that the attacker would guess half the passwords before actually cracking it. Then, based on the rate of hashes, say 1 TH/s, we can directly calculate how long we expect until the attacker cracks the password. From Figure 1 we can see that passwords with less than 65 bits of entropy will get cracked in less than a year, while passwords with 100 bits of entropy can take upwards of 20 billion years.

3 Design

In this section, we present our model for pattern-based password generation.

The CrystalPass application is shown in Figure 2. In the simplest use case, a user provides a pattern in the form of a regular expression in the password pattern box. As they type their pattern, the meter below will dynamically give strength feedback based on the complexity of the pattern. Once the user decides on a pattern they like, they click generate to get their password.

By generating a password this way, CrystalPass gains several advantages. It allows the user to have a single pattern they like and to re-use that pattern, making the individual passwords easier to use. The random generation of the password removes any human baises that would otherwise be hidden in the password. And, since the generation is handled entirely by CrystalPass, it can give accurate entropy feedback on the user's pattern before it is used to create a password.

3.1 Patterns

Patterns are written in the form of regular expressions and are made up of a series of expressions. Each expression represents what type of symbol or string comes next in the password. Such expressions can be words, digits, letters, or punctuation, and can also have various arguments attached to them such as word length, whether to use capital letters, or range of digits.

The general syntax for an expression is as follows: The expression begins with a backslash '\' and then the name of the expression. Consider `\word` as the expression for a single word. This on its own is acceptable and will result in a random word being chosen from the wordlist. However, the user can also add arguments with brackets directly following the expression (*i.e.,* `\word[]`). The valid arguments for `\word` are `length`, `caps`, and `subs`. `length` determines one or more lengths that the generated word can be. This can be in the form of a number (*i.e.,* 5) or a range (*i.e.,* 4-7). `caps` determines whether or not to use random capital letters as `true` or `false`. For words, a user can also capitalize the first or last letter or the word using `begin` or `end`. Note that using random capital letters increases the entropy of the expression greatly, but using `begin` or `end` does not. Lastly, `subs` determines whether or not to use common ascii substitutions such as changing 'a' to '@'. So, a word expression might look like `\word[length=4-7,caps=true]`.

In addition to words, there are 6 other types of expressions that can be incorporated into the pattern. `\digit` generates a string of base ten digits. It can have to arguments `length`, which determines how many digits to generate, as well as the set or range of digits to choose from. The range can be any sequence of digits in the form of 3-8. This range would include all of the digits 3 through 8. The set is given as a string in the form "12345". In this case, the expression would choose from any of the digits given in the string. So, a valid digit expression might look like `\digit[length=2,"12345"]` or `\digit[1-5]`.

`\letter` generates a string of lowercase alphabetical characters. The argument `length` determines how many letters to generate, and the user can also provide a range or set of characters to use. The `caps` argument can be either true or false to use random capital letters.

`\symbol` generates a string of common ascii punctuation, of which there are 32. The user can provide a `length` argument as well as a set of punctuation to use.

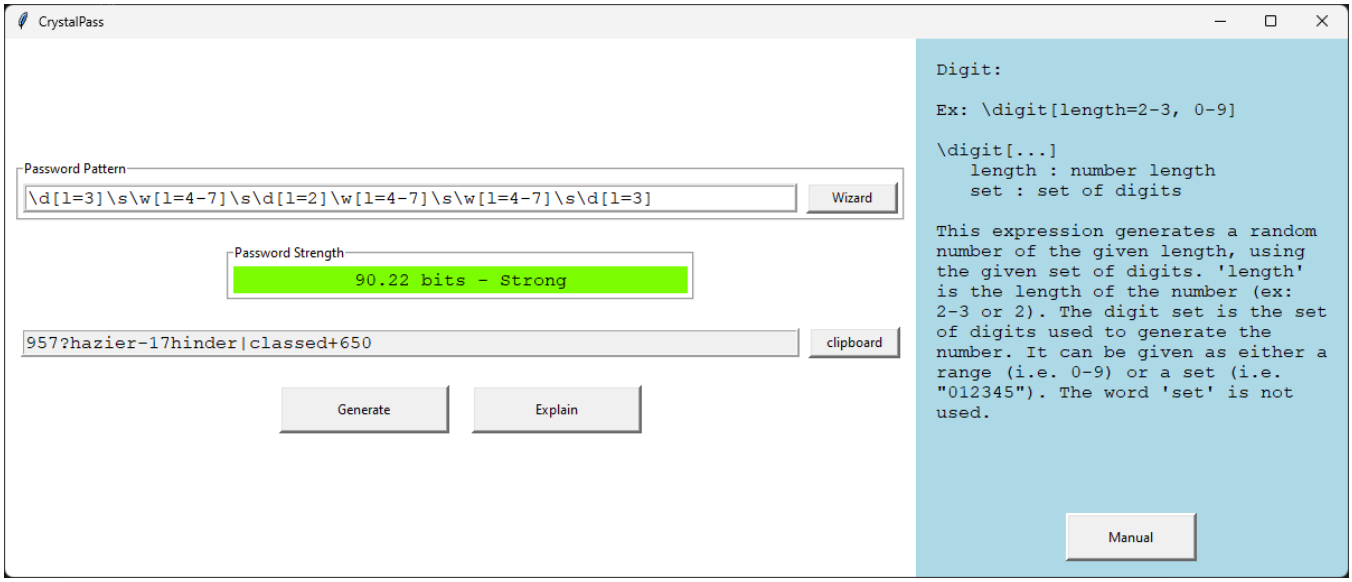

Figure 2: The CrystalPass application window.

`\character` generates a string of digits, letters, and punctuation from the set of all three. As above, the user can provide a `length` as well as a set of ascii characters to use.

The `\named` expression is a little different. Provided for usability, a `\named` expression copies whatever was generated by another expression. Any of the above expressions can also take a `name` argument (*i.e.,* `name=pat1`). This argument simply applies an identifier to the expression so that it can be referenced by a `\named` expression. By default, the referenced expression is generated first, and then the `\named` expression copies what was generated. Alternatively, the `regen` argument can be set to true, causing the `\named` expression to be generated using the same expression and options as whatever is being referenced. A user can use this to save keystrokes when typing their pattern. Additionally, the result of a `\named` expression can be reversed with the `reverse` argument.

Lastly, a `literal` is any string of characters in the pattern that is not a part of any expression. As one might expect, these literals are copied directly into the generated password. Note that literals do not add any entropy to the password and should only be used to decorate a password if desired.

3.1.1 Shorthand

In order to make CrystalPass's expressions easier to type and to fit in the entry box, we implemented a form of shorthand. This means that as long as the expression is not ambiguous, a user does not have to write the whole expression type. For example, `\word` can be shortened to `\w`.

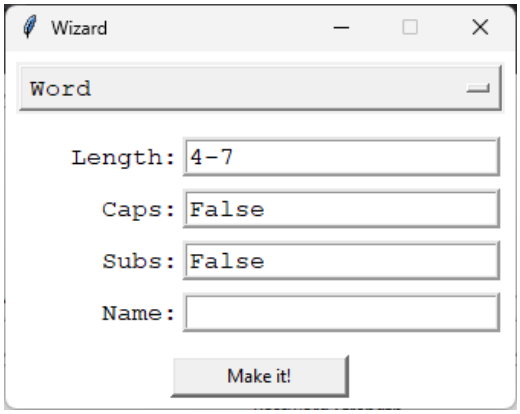

Figure 3: The CrystalPass wizard subwindow.

3.2 Wizard

An important part of the usability of CrystalPass is its wizard (Figure 3). Since most users will likely be unfamiliar with regular expressions and they are not always easy to learn, the wizard provides an easy-to-use interface to craft the next expression. By clicking the "Wizard" button next to the pattern box, the wizard subwindow will pop up. There, the user can choose from a dropdown menu which expression they wish to use. Depending on the expression chosen, different argument options will appear prefilled with the default options. Then, once the user is happy with their expression, they click "Make It!" to append the expression to their pattern. As we will demonstrate in the user study, this feature makes CrystalPass much easier for users to use.

Table 1: Binned entropy feedback.

Entropy	Strength	Time-to-Crack
0-59	Weak/Very Weak	$<$ 1 day
60-69	Reasonable	$<$ 200 days
70-79	Good	18-600 Years
80-99	Strong	62 Million Years
100+	Very Strong	$>$ 20 Billion Years

3.3 Entropy Meter

The entropy meter, located in the middle of CrystalPass's application window, provides dynamic feedback based on the pattern the user has provided. For greater transparency, the feedback is provided in both bits of entropy as well as a binned strength score. The binned strength score is calculated directly from entropy and follows Table 1. Up to 60 bits of entropy, an attacker guessing 1 Trillion hashes per second will guess the password in under a day. These patterns are labelled as "Weak" or "Very Weak" in order to deter users from using them. "Reasonable" passwords are likely to be cracked in less than 220 days, but may be usable for lower-security applications. A "Good" password is useful for most everyday applications though should likely be changed often. Finally, "Strong" and "Very Strong" passwords are likely to stand up to any attack in the near future, taking millions or Billions of years to crack. In addition to the strength rating, the meter will gradually turn from red to green as the pattern becomes stronger. The combination of the above makes it very clear how strong a pattern is to the user before they use it to make passwords.

3.4 Manual

In order to instruct users how to use CrystalPass, we included two forms of Manual. The first is a quick guide which is always visible to the right of the main CrystalPass window. It provides brief instruction on how to write each of the expression types. In order to pack more information into a small sidebar, the quick guide dynamically changes based on what type of expression the user is currently typing. So, if the user begins to type a `\word` expression, the guide will change to show how the expression should be written as well as what arguments it takes.

If the user desires more information on how to construct expressions, how entropy is calculated, or the best practices for password management, we have also provided a full manual. By clicking the "Manual" button, a new window will open containing a left sidebar and a display window. The user can select any of the pages in the sidebar and the corresponding manual page will be displayed. The manual is able to go into much more depth than the quick guide and should be able to answer any questions the user has.

4 Implementation

CrystalPass was written in 1,606 lines of Python code, not including the manual text and the wordlists. Python was chosen as it is easily protable to any operating system which has Python installed. To this end, we only used built-in libraries to make the application as distributable as possible. For instance, we used the Python tkinter package to create the windowed application.

In the remainder of this section, we will describe in detail how various aspects of CrystalPass were implemented.

4.1 Pattern Parsing

The pattern is parsed in a single pass from left to right. The parser begins by searching for the first expression, which is denoted by a \, followed by any valid expression type. For both expressions and arguments, CrystalPass performs longest substring matching. This means that so long as the substring is not ambiguous, shorthad versions can be used. For example, `\letter` can be written out in full, or simply as `\l`. Since it is the only expression to start with the letter '`l`', it can be correctly parsed as the `\letter` expression. The same is true for arguments: `l=2` would be parsed as `length=2`, since that is the only argument that starts with an '`l`'.

Once the expression type has been parsed, CrystalPass identifies the argument bounds '[' and ']'. It then parses each of the arguments between those bounds. It should be noted that spaces withing the argument bounds are ignored during parsing, so '`length = 3 - 5`' is the same as '`length=3-5`'.

Parsing occurs based on two actions: the user types a new character in the pattern box, or the user presses the generate button. The former is to update both the feedback meter and the quick guide. This simply makes the dynamic feedback snappier and easier to use. The latter makes more sense as the pattern needs to be used to generate a password. Note, if there are any parsing errors such as incorrect syntax, an error is raised in the form of a pop-up window, but only in the case where the user has attampted to generate the password. This is done for usability concerns.

In the case that an expression is given a `name` argument, that expression is stored in a map where the key is the name. When the name is referenced in a `\named` expression, the name is simply dereferenced in the map.

Lastly, as the pattern is parsed, it is stored as an array of data structures, where each array element is its own expression. This will allow for easy entropy calculation and password generation in the following steps.

4.2 Password Generation

We will explain the password generation process first. Since the pattern is stored in an array, all that remains is to traverse the array and generate each expression in turn. In most cases

this simply means choosing random symbols from the set defined in the expression, then applying any necessary modifications. For any random choice CrystalPass makes, it uses the Python secrets package, which uses the best random number generator available to the system. This is done to reduce exploitable distributions in the generated passwords.

`\symbol`, `\digit`, and `\character`, are defined by the set of symbols given in the arguments and their length. For these expressions, CrystalPass chooses a random symbol in the set for length number of times.

For `\letter`, capital letters can be used. So, it is generated based on the set of lowercase letters for the given length, and then each letter has a 50% change to be shifted to an uppercase letter.

For any of the above expressions, if the `length` argument is provided as a range, a random length is chosen from the range first, then the expression is generated using that length.

`\word` expressions can have a word length given as a range, as well as an addlist and a blocklist that modify the wordlist. At generation time, CrystalPass creates a single array of all the words in the wordlist that are of the given lengths and follow the add and blocklists. CrystalPass then randomly chooses a word from the list with uniform distribution. If ascii substitutions are used, CrystalPass consults a map which maps lowercase letters to common ascii substitutions. For each letter that can be substituted, the substitution is used 50% of the time. After this, if capital letters are used, the remaining letters are made uppercase with 50% chance.

If a `\named` expression is found, CrystalPass looks in the name map to locate the referenced expression. Because of this, an expression before the `\named` expression must have a `name` argument with the name in question. By default, the `\named` expression simply copies what was generated by the referenced expression, though it can either reverse the generated expression or regenerate the expression altogether.

Any string found in the pattern that is not a part of an expression is treated as a literal and copied directly into the resulting password.

4.3 Entropy Calculation

Our entropy calculation is based on the attacker described in §5.1. This is a strong attacker who knows what pattern was used to generate the password.

Entropy calculation is performed on the pattern that has been parsed and stored in an array. The calculation is fairly simple: for each element, find the entropy that it adds to the pattern, and add it to a summation. The entropy that an expression adds is based pruely on the number of symbols it can choose from (*i.e.,* words, letters, digits, *etc.*). In the case that more than one symbol is generated for one expression, the set of symbols is raised to the power of the number of symbols. The log scale is then performed at the expression level, and the logs are added together.

$$E = \sum_{x \in X} \log_2(|S_x|^{L_x}) \text{ bits} \tag{3}$$

In equation 3, the entropy is shown as the summation over all expressions, where S_x is the set of symbols in the expression, and L_x is the number of symbols to be generated. Note that this would be the same as multiplying the number of combinations, and then taking the log. The special cases are described below.

For `\word` expressions, the set of symbols is the final wordlist, after the correct word lengths have been chosen and the addlist and blocklist have been applied. If capitals or ascii substitutions are used, these possible combinations are accounted for when calculating the size of the symbol set (S_x).

If an expression length is defined by a range, the average of the range is taken to be the length of the expression.

A `\named` expression in the default mode of copying what was generated by the referenced expression will not add any entropy. The same is true if it only reverses what was generated. However, if it regenerates the expression, it has the same entropy as the referenced expression.

4.4 Entropy Accuracy

Because we can accurately calculate the size of the symbol set for each expression as well as the expected length of each expression, we know the number of possible passwords that could be generated by a given pattern. In addition, since we use the strongest random number generator available on a system, we assume that there is no percieved bias in the choices CrystalPass makes. Thus, each password will be generated with equal likelyhood. Taking this into account, we know the expected number of guesses for an attacker to guess any one password (half the total number of possible passwords). Since our entropy calculation follows directly from this, we prove that our entropy feedback is correct for an attacker who is aware of the pattern used.

5 Evaluation

In this section, we will discuss the evaluation of CrystalPass. We will first present our threat model, then discuss various aspects of generated password strength and the entropy meter. Fianlly, we will present a small scare user study.

5.1 Threat Model

In order to ensure the strength of passwords generated by CrystalPass, we compare against a very strong attacker. Namely, the attacker performs an offline attack against the hashed and salted password(s) and knows what pattern was used to generate them. We note that because of the random generation, knowing the pattern that was used is the most information ana

attacker can have short of knowing the actual password. We then assume the reasonable compute capability of 1 TH/s or one trillion guesses per second.

For this work, we do not consider direct attacks against CrystalPass during password generation. We only consider password cracking attacks once the generated password is in use. We also assume that the passwords are properly hashed and salted.

5.2 Password Strength

The first major question of this work is "Is it possible to create strong passwords using patterns?" To this, the answer is a resounding yes, but it can take some work. For instance, a password that is simply 4 words back-to-back will have an entropy around 63 bits. Since we assume the attacker knows this pattern, it will only take them a few days to guess the password. However, if we craft a password following the pattern:

```
\d[l=2]\s\w[l=4-7]\s\w[l=4-7]
\s\w[l=4-7]\s\w[l=4-7]\s\d[l=2]
```

we get an entropy of over 96 bits. A password of this strength would take tens of millions of years to crack, even though the attacker knows the pattern. And this pattern could be made even stronger if random capital letters were used.

We do find that to get the highest entropy with the shortest generated password, nothing beats purely random passwords. CrystalPass can support this with a single expression: `\c[l=20]`. This expression generates a password of random letters, numbers, and punctuation, of length 20, and achieves an entropy of over 131 bits. The tradeoff is that these passwords are less usable.

We have found that a good strategy for building strong patterns is to primarily use words as they have a high entropy and are easier to remember. Then, to make the password stronger as well as easier to read, we sprinkle some punctuation or numbers between the words.

5.3 Entropy Meter Comparison

Most password feedback meters use some form of entropy analysis in combination with a password blocklist. That is, they provide an entropy analysis based on a brute-force attacker, and then penalize the entropy score if the password appears in the blocklist. The brute-force entropy analysis accounts for the different sets of symbols that a password contains. So, a password of length 20 that only contains upper and lowercase letters would be weaker than a password of the same length containing those letters in addition to punctuation. This is a consequence of equation 2 in §2.2. This kind of entropy analysis does not take into account words, which should be treated as indivudual symbols. As a result, typical entropy feedback meters grossly overestimate the strength of passwords, providing users with a false sense of security.

CrystalPass has the advantage of being in control of password generation, namely ensuring that the password distribution is uniform. Based on the pattern provided, it is able to count up the exact number of possible passwords, and calculate entropy directly from that. This means that any password generated through CrystalPass will have an accurate password feedback based on our attacker.

Considering our attacker is very strong, CrystalPass holds its passwords to a higher standard than most dictionary or brute-force attacks. Because of this, a strongly rated pattern will provide a password that can stand up to any weaker attacks.

5.4 Pattern Re-use

The purpose of CrystalPass is for a user to have a single pattern that they become familiar with as they re-use it to generate several passwords. In light of this, we study the strength of the generated passwords in the face of pattern re-use.

For this work, we assume that the password hashes are properly salted. In this case, an attacker must re-hash their guesses for each entry in the database. It makes no difference if the attacker tries to guess the passwords in turn or all at once. So, due to the slowdown caused by salt, the attacker does not gain any advantage when trying to guess any one of many passwords generated from the same pattern.

Building on the passwords' strength even against an attacker who knows the pattern that was used, it is reasonable for the user to store a sufficiently strong pattern in plaintext for future use. We will touch on this more in §6.2, where we lay out ideas for improving the re-usability of patterns.

5.5 User Study

We now present a user study conducted on a small group of friends and family of the authors. We know that the group contains both users with extensive computer experience as well as those who do not have much experience. Out of about 12 people who were asked, 6 ended up completing the study ($n = 6$). The purpose of this user study is to determine the usability of CrystalPass to the average user, as well as the users' preference to using CrystalPass over other password generation methods.

The user study was carried out independently by the users in their own time. The users were asked to install the CrystalPass application and use it to create a password to their liking. They were then questioned based on their current practices, the usability of CrystalPass, and their preference to use it in the future.

5.5.1 User Practices

The first section of the questionair asked users how they currently generate passwords and how they manage them. Key to this work, 50% (3/6) of users mentioned that they used a single password that they changed slightly for different accounts or re-used outright. In addition, 5/6 users thought that their passwords were not strong enough. This is clear motivation for an application like CrystalPass, which provides users with usable yet secure passwords.

5.5.2 Usability

The usability of CrystalPass brought up several concerns to first-time users. Namely, 5/6 users agreed that writing their own expressions and patterns was too difficult. One user noted that "It wasn't obvious to me that I should actually include the brackets in the options." This indicates that the expression syntax has a learning curve to it or may not be obvious how to use. Thankfully, all users agreed that the wizard was much easier to use. So, despite some concerns, there is a way for users to make expressions that is easy to use.

In addition to the struggle of using expressions for the first time, users appeared to struggle to make usable patterns. We note that while `\word` was the most commonly used expression, few users used it more than once, failing to take advantage of its usability. Instead, many users opted for random letters or symbols as the source of their entropy. In essence, users preferred more dense passwords - shorter, but also harder to remember. For instance, one user generated `oCeanS@<~|TYnpsszx`, which is a single word followed by many random characters. Despite the fact that strong patterns can be made with words, users either did not trust them or did not realise that words could be useful to make strong, usable patterns.

One user suggested that the ascii substitutions option should be modified so that is is not a simple true or false, but that the user could determine what types of substitutions are made. In their comment they suggested that a user could "choose to add one or two symbols per word, or maybe just turn vowels into symbols."

Another user noted that the `\named` expression was too complicated. They struggeled to understand the purpose of it or how it works. The expression likely needs a more in-depth manual page with more examples.

Lastly, we found it humerous that a user commented "Does not know any words with more than 21 letters." While this is true, we note that CrystalPass uses the standard dictionary that ships with Ubuntu 22.04. In addition, for users who are willing to dig a little, we provide an addlist and blocking that can be modified to add or remove words from the wordlist.

5.5.3 User Preference

While overall the feedback for CrystalPass was positive and the users largely enjoyed the experience, only 3/6 expressed a desire to use CrystalPass in the future. For one user, they already used a password manager and reported that their passwords were strong. For the others, the complexity of expressions and patterns were likely a deterrent.

That said, one user reported that CrystalPass is "Way better than the stupid Apple password generator," likely referring to random passwords, and another said, "I rate it 5/7, very nice!" From this we can draw that CrystalPass is a step in the right direction for password generation, it just has some usability issues holding it back.

Finally, one user reported "I mean of course I'm gonna use it, I'm your wife, that's like my job," which was very cryptic and we are still trying to figure out what that means.

6 Discussion

In this section, we will discuss some limitations of CrystalPass, as well as avenues for future work.

6.1 Limitations

While we present CrystalPass as a complete work, there are still some issues that it may face in practice.

6.1.1 Possible Vulnerabilities

CrystalPass does not make any guarantees on the security of the application against direct attacks. For instance, an attacker could easily read the application's memory to skim a user's newly generated password. It also still falls to the user to store their new password properly using a password manager. Since CrystalPass is coded in Python, we did not make an attempt at creating a password manager replacement. Such a feat is likely infeasible using generic Python, and would not be portable to mobile devices.

We also assume that there is not bias in the password generation process since we use the Python secrets package. In reality, the password generator is only as good as the best the host hardware can provide. If an attacker were to find a bias in the hardware, they would also be able to perform a stronger attack against CrystalPass.

6.1.2 Making Strong Patterns

In this work, we found that while making strong patterns is possible, there is a bit of a learning curve to it. We demonstrated in §5.5.2 that users struggle to find a balance between strong patterns and usable ones. On the other hand, we did not want to provide an example pattern directly for fear that users would collectively use that single pattern.

In addition, we made the requirement for a password to be rated as strong to be very high (80+ bits of entropy in Table 1). We note that while users struggled to hit this bar, we are also unable to lower it in good conscience. That said, it is clear that the majority of users will need some form of assisntance or hints to get the full utility out of CrystalPass.

6.1.3 User Study

The user study has its own share of limitations. First, the user base consists of $n = 6$ users who directly knew the author as they were friends and family. While this provides the advantage of improved engagement, it also introduces a bias where users are less likely to be critical or unknowingly rate it higher than they otherwise would. We make note of this and instead treat the user study as a pilot from which we can improve CrystalPass for a future, more rigorous user study.

We also notice that the use of Python was a high entry bar for some users who were unfamiliar with the language. Some potential users were uncomfortable installing a tool that they had never heard of, and thus did not take part in the study. While it would reduce the portability of CrystalPass, this does show the benefit of shiping pre-compiled code.

6.2 Future Work

Moving forward, we have three major improvements we would like to make to CrystalPass.

First, users need an easier, more interactive way to make patterns. The built-in wizard is great for first time users, but it only appends the expression to the end of the current pattern. Users will likely struggle then if they want to insert an expression somewhere in the middle of their pattern, or want to edit an existing one. Admittedly, this kind of UI design may be beyond me, but it is worth the try.

Second, CrystalPass needs a way to nudge users towards patterns that are both strong and usable. The entropy meter provided was enough to get users to try to make strong patterns, but they often sacrificed usability. We see it as future work to solve this problem without providing a blanket example pattern.

And lastly, we want to build-in a way to save a user's pattern for future work. While a strong enough pattern could be stored in plaintext, we do not want this to be the final solution. We want our future work in this area to provide a secure method of storing the password, likely encryption, that makes it difficult for an attacker to directly read the user's pattern. We expect this to be an adventure in encryption practices.

7 Related Work

In this section, we will discuss some of the tools available to users as well as password strength enforcement and feedback seen in related works.

7.1 Password Cracking Attacks

Pass2Edit [10] is a first-look study into attacks against password tweaking, where a user only slightly changes their passwords for different accounts. If a password is known for a user's account, they attempt to guess that user's passwords on another account by applying common changes to the known password. They demonstrate that in as few as 100 guesses, their success rate is as high as 24%.

Nisenoff *et al.* [5] presents a full attack against re-used and tweaked passwords created by university students. For example, if a user has a password such as `321movies123` for their Netflix account, they may only slightly change the password to `321mail123` for their email account. The authors show that these slightly changed passwords become easy to guess for a targeted user across many of their accounts. Their success rate is as high as 32% for only students in the same university as the authors.

Another common attack is the dictionary attack. A recent example of this is put forward by Xu *et al.* [11]. They take the basic dictionary attack which guesses random words and common tweaks, and augment it to guess based on a known distribution of common substrings, such as `4ever`. Such attacks are much stronger than current feedback meters can account for because they take advantage of unseen patterns in the password distribution.

CrystalPass will be able to defend against each of these attacks. Since the password generation is done randomly, there is no human bias to take advantage of. Also, since the size of the wordlist is taken into account during entropy calculation, we can prove that a strongly rated password will withstand a dictionary attack. And lastly, since we assume the attacker knows the pattern during entropy calculation, we know that tha strongly rated password will withstand a "pattern reuse" attack.

7.2 Password Strength

Sahin *et al.* [6] investigates the use of password policies in websites, and finds that most policies are either used because they are standard practice or relaxed due to usability concerns. Tan *et al.* [8] studied the effects of how password requirements encourage users to make stronger passwords. They find that requirements such as minimum-strength and minimum-length are best for high-security applications, though their strength measurement is based on a neural network, and may be infeasible in practice. Tan *et al.* also studies the impact of blocklists, but these are usually primitive in design and are unable to block passwords that do not exist in their database.

Password strength meters are commonly used to inform users of their password strength while they are creating it in the hopes that it will nudge users to make stronger passwords. Wang *et al.* [9] studies these meters and shows that they may be inaccurate at best. In some cases, a meter could be fooled

into telling a user that `password123_` is a strong password, despite common knowledge dictating that it is not. To some extent, this is because the meter cannot take into account that a common word is being used as the core of the password.

Again, since the entropy feedback is calculated based on the pattern, and the password is randomly generated, we can prove that CrystalPass's entropy feedback is accurate.

7.3 Password Generation

Most password generators available to users today simply create random strings of characters. While it can be proven that these are strong against every attack, they are also very difficult for users to use due to their randomness and difficulty to remember.

There has been some recent work into the creation of usable passwords that are still strong. AutoPass [3] presents an end-to-end password generator and manager intended to span multiple platforms. AutoPass generates a completely random password for each site a user needs to create an account for. The tool maintains security by splitting up the password and its salt. In order to login to an account, AutoPass needs the password from the user's device, and the salt from the AutoPass server. The password is hashed with the salt and then used to login to the user's account. While in practice this work promises to be seamless, it requires a user to switch entirely to using AutoPass.

Alphapwd [7] takes a more fun approach to password generation. The authors begin with a very simple combination of characters, then overlay those characters' shapes onto a keyboard. The password is then the series of keys pressed in writing out the shapes of those characters. This strategy creates a relatively easy to remember password as the user only needs to remember a few letters, yet strong since each character expands to many key-presses. While novel and interesting, the creation of passwords in this way will lead to a set of similar substrings. These substrings can then be used in a type of dictionary attack to guess passwords. In their paper, the authors do test the security of their passwords, but the results are not promising. They trained a machine learning model on large datasets of leaked passwords and then attempted to guess the passwords generated by Alphapwd. 40% of their generated passwords were guessed in less than 10^{16} guesses. These passwords are likely not strong enough to withstand the attacks put forward by modern attackers.

Lastly, Glory *et al.* [4] puts forward a method of password generation based somewhat on user input. The user is asked to input 5 texts and 2 numbers which are then modified with common substitutions (*i.e.,*'a' and '@'). The main issue here is that this will still likely be weak to dictionary and tweaking attacks. The user inputs will still have the same unseen patterns that were used to crack passwords in [11]. In addition, the authors' entropy analysis shows that the passwords only have between 47 and 88 bits of entropy. While the upper end of that will withstand attacks for some time, 47 bits is far too weak to use as a password.

CrystalPass will be able to outperform each of the above password generators. We do not require that users use any kind of end-to-end solution, we only generate strong passwords. We also allow users to choose a pattern that they will find easier to use while still giving them strength guarantees based on the pattern.

8 Conclusion

In this work, we aim to tackle the problem of users being required to generate unique, strong passwords for each of their accounts. We introduce CrystalPass, a pattern-based password generator, which allows users to create strong and usable passwords while re-using a common pattern. We demonstrate the security of such patterns through a proven accurate entropy analysis compared to a strong attacker. Lastly, we demsontrate the usability of our tool through a user study.

References

[1] ';–have i been pwned? https://haveibeenpwned.com/Passwords, 2024.

[2] Password Entropy: The Value of Unpredictable Passwords. https://www.okta.com/identity-101/password-entropy/, 2024.

[3] Fatma Al Maqbali and Chris J Mitchell. Autopass: An automatic password generator. In *2017 International Carnahan Conference on Security Technology (ICCST)*, pages 1–6. IEEE, 2017.

[4] Farhana Zaman Glory, Atif Ul Aftab, Olivier Tremblay-Savard, and Noman Mohammed. Strong password generation based on user inputs. In *2019 IEEE 10th annual information technology, electronics and mobile communication conference (IEMCON)*, pages 0416–0423. IEEE, 2019.

[5] Alexandra Nisenoff, Maximilian Golla, Miranda Wei, Juliette Hainline, Hayley Szymanek, Annika Braun, Annika Hildebrandt, Blair Christensen, David Langenberg, and Blase Ur. A {Two-Decade} retrospective analysis of a university's vulnerability to attacks exploiting reused passwords. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5127–5144, 2023.

[6] Sena Sahin, Suood Al Roomi, Tara Poteat, and Frank Li. Investigating the password policy practices of website administrators. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 552–569. IEEE, 2023.

[7] Jianhua Song, Degang Wang, Zhongyue Yun, and Xiao Han. Alphapwd: A password generation strategy based on mnemonic shape. *IEEE Access*, 7:119052–119059, 2019.

[8] Joshua Tan, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Practical recommendations for stronger, more usable passwords combining minimum-strength, minimum-length, and blocklist requirements. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1407–1426, 2020.

[9] Ding Wang, Xuan Shan, Qiying Dong, Yaosheng Shen, and Chunfu Jia. No single silver bullet: Measuring the accuracy of password strength meters. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 947–964, 2023.

[10] Ding Wang, Yunkai Zou, Yuan-An Xiao, Siqi Ma, and Xiaofeng Chen. {Pass2Edit}: A {Multi-Step} generative model for guessing edited passwords. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 983–1000, 2023.

[11] Ming Xu, Chuanwang Wang, Jitao Yu, Junjie Zhang, Kai Zhang, and Weili Han. Chunk-level password guessing: Towards modeling refined password composition representations. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 5–20, 2021.