# Closing the Gap Between Memorable and Secure Passwords

Matthew Jones

12 December 2018


## Abstract

A good password has two key features: it is difficult to guess and therefore generally difficult to crack, and the user can remember it. The first feature faces issues from user-generated passwords, since people tend to gravitate towards common nouns (such as hobbies or names) or simple patterns (such as 123456 or qwerty). Common solution to this problem, such as forcing users to include capital letters, digits, or other symbols are generally unsuccessful, because people tend to work around these rules in common ways (i.e. putting special symbols at the end of the password). As a result, smart password crackers can work around this condition, and the passwords are harder to remember.

The second feature, memorability, is important because otherwise users may need to change their password each time they log in or somehow record their password, both of which are detrimental to security, or users may simply be locked out of their own accounts. This issue is often worse with computer-generated passwords, which have a strong randomness and are secure, but may be very difficult for people to remember.

This paper examines how to bridge these features, creating passwords which are both memorable and secure. Specifically, it examines how successfully computer-generated passwords can satisfy both features, maintaining the randomness of passwords while using human-friendly strategies, such as using short sequences of common words instead of long sequences of characters.

## Table of Contents

# Introduction

The concept of passwords is much older than computers, and is not limited to the fields of technology and cybersecurity. The concept of a password was mentioned at least as early as 146 BC, in *The Histories*, by the greek philosopher Polybius [1]. However, the invention of the modern computer, with the notion of a password for a computer system, forced a shift in the way that people thought about passwords and their security. For a single guard standing at a locked door, a simple word is sufficient as a password. First, if a person is guessing, that person can only guess passwords or passphrases so quickly, and a random word might be enough that if somebody were to guess words exhaustively for a whole day, they still wouldn't guess the password correctly. Second, a guard would probably be able to see who is telling them the password, so they would be able to recognize if a person is just guessing, and could stop responding to that person. Third, if the password is complicated or obscure, a guard may be able to make a well-reasoned decision about whether the person actually knows the password and is simply remembering imperfectly, or if they are guessing. However, for passwords in the realm of computers, all of these become issues.

The first issue is the number of guesses that a single party could make. While a person or group of people could only guess a relatively small number of passwords in a short amount of time, computers are much, much faster than people. In fact, a team of password-cracking experts at Stricture Consulting Group had a system which utilized 25 graphics cards and was reportedly able to guess 350 billion passwords per second encrypted with the NTLM encryption used in Windows systems [2]. This was reported back in 2012. As a result, modern passwords need to be much, much more secure than a single random word which would have sufficed back in the days of the Roman Empire.

The second issue deals with how many times a person or entity is allowed to guess a password. If a human is on the receiving end of the guess, they can make an intelligent decision on when too many guesses have occurred, and stop responding if that is the case. However, it is not so easy for a computer to do the same task. Even with a solution to this problem, often this problem isn't even the significant issue anyways. In many cases, a password cracker has already gained access to some information about the password, and simply has to try to solve it. The solution here is password hashing, in which a password is first pushed through a one-way algorithm, and only the resulting hash is remembered by the system. In this case, a hacker can only gain access to the hash, and then must figure out how to replicate the hash by guessing. Therefore, the best way to protect the password, as well as using a strong hashing algorithm, is to use a hard-to-guess password, even at the guessing rate of a computer.

The third issue is in direct conflict with the hashing solution to previous issue. Anybody who has used a computer and needed a password for a website or a service has probably forgotten a password at some point, and a study at Rutgers University suggests that if people use a password as infrequently as once every 4 days, they recall the password correctly on the first try less than 10 percent of the time [3]. The only real solution to this, besides writing down passwords, would be to let our passwords be less perfect, and grant authorization if users get "close enough" to the actual password. However, good hashing algorithms are irreversible: given the output, it's practically impossible to get the input. As a result, a slight modification to the input can produce a drastically different output, so the idea of "close enough" breaks down when hashing is used. As a result, hashing makes it important that passwords are not only secure, but memorable.

This is the problem this paper addresses, maintaining security while also achieving memorability. The key solution proposed is to redesign the system behind the passwords, such that the building blocks of passwords are words, not characters, since humans can often remember a short sequence of words better than a long sequence of characters [4]. Doing this with a large set of words can yield sufficiently random passwords, with a much higher recall rate among users.

## To The Community

Asking why this overlap of memorability and security is important is the same as asking why passwords are necessary in the first place. Passwords are designed for the purposes of authorization, whether that means verifying that the person posting information under a name is actually that person, protecting the privacy of sensitive information which a user stores on a computer, or limiting who has access to sensitive information. The danger of an insecure password is the danger of risking the soundness of any of these forms of authorization. Without password security, passwords simply aren't effective, which makes them significantly less useful for the important purposes they serve.

As a user, a password should be something that you can wield effectively. It should be easy for you to use, but it should also protect your data effectively. The goal of creating such passwords is a goal that anybody who uses a computer should reach for, regardless of whether they are a user or a software engineer.

In that vein, it is worth noting that such password systems are not unachievable. However, they are nontrivial to create, and the burden of designing and using such systems should be shared between users and designers as both have contributed to this issue in the past. Users create passwords using the same common approaches, and use the same heuristics to make easier and, inherently, less secure passwords. They think primarily about how the password is used by a person, when they should orient their thinking as to how the password is attacked by a computer with a clever program. Similarly, designers tend to use the same naive approaches to try and force users to make secure passwords. They think primarily about how the password is used by a computer, and they fail to understand how a person would approach password creation under the restrictions the designers impose. Both parties think about the problem with respect to their own area and not the other environments around the password, and, as a result, both parties contribute to the weaknesses of passwords.

This paper specifically looks at how a software designer would create such a system, but it contains elements which are useful to both parties. For a designer, this paper will highlight why the current methods for password spaces are less-than-effective, and will discuss better approaches. As a designer, this paper should be read as the introduction of a new tool, that may be slightly more difficult to implement but will allow users to more easily use the password system without sacrificing security. At the very least, it should help a designer to think about how the average user would work with their system, and help highlight weaknesses with approaches to password systems. For a user, this paper will highlight some of the key issues users make in password creation, and will try to offer other approaches to password creation. As a user, this paper should be read as a critique of password creation approaches, and the paper should be a useful tool for critical analysis and revision of the reader's methodology for passwords. At the very least, it should assist a user in realizing that some of the password-building techniques they use are more common than they realize, help the user to understand the crucial gaps these techniques leave in password security, and force the reader to think more carefully about how they should design their passwords. In either case, the concepts in this paper are relevant and important to anybody who uses passwords on a computer or on the Internet.

# Preliminaries - Theory and Motivation

## The Security of Passwords: The Concept of Entropy

Formally, what makes passwords secure, and why are people bad at making secure passwords? To answer this question, it's best to introduce the idea of Shannon entropy [5], which will often be referred to as just 'entropy' in this paper. Shannon entropy is a value which can best be interpreted as an answer to the question, "How many bits would I need to encode a piece of data". For example, a coin flip would have entropy 1, since the result can be stored in a single bit, as heads or tails. Rolling a single die would have entropy $\log_2 6$. More formally, if one has the optimal encoding for events so that if they searched a binary tree for the events one digit at a time the expected time of our search would be minimized, that expected time would be the Shannon entropy. If one event occurs with a relatively high probability, it would have a one- or two-digit encoding, so a binary tree search would find it very quickly, and it has a relatively low contribution to entropy, whereas if an event occurs with very low probability, it has a long encoding and is very deep in the tree, but has a small weighted contribution to entropy due to its low probability. In general, if there is an atomic (disjoint and complete) set of events $\{e_1, e_2, ..., e_n\}$ with probabilities $p_i$ such that $\sum_{i=1}^{n}(p_i) = 1$, then the Shannon entropy of the set is calculated by

$$E(e_1, e_2, ..., e_n) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

Note that it is not particularly reasonable to calculate the actual entropy for a set of passwords, because it is nearly impossible to collect actual data on the frequency of each individual password across all time. It is better, then, to note a few properties about entropy, and use them as heuristics for evaluating entropy:

- Entropy is maximized across a set of events when all events occur with equal probability. As probabilities tend to move away from $\frac{1}{n}$ in either direction, entropy tends to decrease.
- Entropy is generally increased by adding more events. Specifically, if each event occurs with probability $\frac{1}{n}$, then entropy is equal to $\log_2 n$, which is increasing with $n$.

The difficulty of guessing a password is closely related to the entropy of the space of passwords [6, 12]. If there are more possible passwords then it is more difficult to guess a password, and there are a larger number of events and therefore a larger entropy. If there are a few incredibly common passwords, though, then password cracking can be very successful by focusing on these passwords first, and entropy is lower due to the fact that password probabilities deviate from the $\frac{1}{n}$ average. To examine why this is the case, give each password the encoding and build the binary tree which would give us the optimal search time as atomic events in Shannon entropy. It is then possible to guess passwords in an optimal order by moving from smallest to largest encoding. The weighted average length of the encodings is the Shannon entropy, so the time to guess a password is proportional to the number of these encodings, which is approximately 2 to the power of the entropy. As a result, if a password space has Shannon entropy $E$, the time to crack a password in that space is approximately proportional to $2^E$. In practice, password crackers generally never have the entropy or optimal encodings, but they can achieve a similar optimal efficiency by guessing more common passwords first and less common passwords last. As a result, increasing entropy is an exponentially effective approach to making password cracking more difficult.

However, people are generally bad at making passwords that achieve high entropy. When choosing passwords, people will use passwords that satisfy requirements put before them and require as little effort as possible to remember [7]. Specifically, people use a number of heuristics to make passwords easy to remember, including reusing passwords, using words or dates related to their life, replacing letters with similar numbers or symbols (such as $ for S, @ for a, or

0 for O), or putting certain characters in specific places, such as captial letters at the start of the password or digits and symbols at the end of the password [8, 9, 10]. The issues of common password-creation methods, common character substitutions, and common positioning are detrimental to security because when many people are using the same approach to the passwords, it becomes more likely that passwords matching those approaches will be used, increasing the probability of those passwords and lowering the entropy of the passwords. As a result, people create insecure passwords when no restrictions are enforced [10], and when restrictions such as symbol usage and capitalization are enforced, passwords often become only marginally more secure.

### The Memorability of Passwords and the Dangers of Changing Them

While security is certainly the biggest and most important feature of passwords, the ability of a user to remember their passwords is also incredibly significant. The most obvious reason for this is simply ease of use. If a person has trouble remembering their password, authentication can be very burdensome. One possible example of this could be that this is the primary difficulty of the common solution services use to prevent brute-force guessing. To prevent attackers from guessing usernames and passwords iteratively, services may lock the login process for a user account for a short time after a few incorrect attempts. While this is effective in its goal, it can become quite bothersome for a user who mistyped or mistook their password, and is then locked out of a service at the exact time they were trying to access it. This will happen more frequently as passwords become less memorable, and will make it generally more frustrating for users to use the system. As a software designer trying to attract people to a product, this can be detrimental. The other option is that a user may have to reset their password each time they need to log in. This can be equally bothersome to being locked out of a system.

Also, the act of changing a password is not necessarily beneficial to security. If a user is creating their own passwords, they will often change the password as little as possible, such that the new password is easy to remember given that they could recall their old password [11]. In many cases this means incrementing a number by 1, changing the case of a letter, or only changing a symbol. Password system designers use similarity testing to prevent this, but people will still try to find a minimum-effort workaround. In fact, a study at UNC found that when people are forced to change their password multiple times due to expiration, they will often change them in the same way [11]. The ramifications of this are that while regular password changes are effective, they are considerably less effective than designers expect them to be, and if an attacker can break a user's passwords both before and after a password change due to expiration, they have a strong guide for guessing that user's passwords after future changes as well.

At this point it is worth noting that other methods exist for remembering passwords than just mental memory. For example, passwords can be simply written on paper in a secure location. While this may be sufficient in some cases, in others it is simply to risky to have a physical copy of the password somewhere it could be found by another person or by a webcam which has been compromised. Even if these are not the case, many corporate policies prevent employees from recording login credentials, which would prevent this solution. In any case, it may be unwise or unsafe to keep a physical copy of a password. Another common tool is a password manager, which will handle randomly generating passwords and keeping track of them. This tool has a lot of positives, such as the fact that passwords are secure and many managers can periodically remind you to change your passwords. However, these also come with the obvious risk that all of your passwords are in a single location, so any attacker who gains access to your password manager immediately has access to all of your passwords. This issue also comes with the common technique of using the same password or similar passwords across services or websites, where an attacker gaining access to one account can compromise all of the other accounts as well. As a result, while methods exist for remembering passwords without actually keeping them memorized, these methods are generally risky for their own reasons.

# Applications - Solutions to the Problem

## Proposed Solutions

The general approach to closing the gap between security and memorability is to use a sequence of items which people can remember well. The approach which will specifically be discussed in this paper is to use words as the building blocks which make up a password, rather than characters. The motivation behind this is that since people use words on a very regular basis, people tend to be good at remembering short sequences of words. Also, a common trick for remembering sequence-of-characters passwords is to use a mnemonic, so why not simply remember words to begin with?

The earliest edition of this idea was from Stanley Kurzban, in 1985 at IBM [13]. Kurzban's idea, called 'Easily Remembered Passphrases', had 4 lists: a list of adjectives, a list of actors (which are subjects of a sentence), a list of verbs, and a list of things (which are predictates of a sentence). Each list contains one hundred items, such that each item in a single list had a different 3-letter identifier. For example, in the actors column, admirals had the identifier 'adm', accounts had the identifier 'acc', ants had the identifier 'ant', and so on. By using three lists, for example the actors, verbs, and things lists, a three-word phrase can be built from each of the triplets where one word is taken from each list. As a result, this would give 1,000,000 possible password phrases, which could be potentially identified by a number (between 0 and 999,999), an abbreviation (such as admaddacc for 'Admirals Add Accounts'), or the phrase itself. Kurzban noted that this could be implemenented in any of the three ways, for example the passwords themselves could be abbreviations and the phrases could be mnemonics for the passwords. The obvious weakness of this system is that at this time, 1,000,000 password combinations is a relatively small number, and the average desktop computer could break that very quickly. However, he noted that more than a million password phrases is possible by combining more than 3 instances of lists, and their isn't a theoretical minimum to the number of times lists could be used to increase the number of combinations.

More recent works tend to be inspired by an XKCD cartoon [14]. In the cartoon, the proposed solution is to transform a random 44-bit sequence, which would have an entropy of 44 from a total of about 17.6 trillion combinations, into 4 words which would compose the password. This approach, published over 25 years after Kurzban's approach, doubles the entropy of the one million combinations Kurzban proposed.

One paper to take inspiration from the cartoon was *Correct Horse Battery Staple: Exploring the Usability of System-Assigned Passphrases* [15], named after the example passphrase in the cartoon. The results in the paper come from a study to determine how well people remembered randomized passwords from different sets and passphrases (random sequences of words) of varying lengths. The results from this paper were not overwhelmingly positive, but the passwords and passphrases in this paper were not secure enough either. Specifically, the randomly-generated passwords tested as a baseline were random 5-character and 6-character passwords from all symbols, and random 8-letter pronounceable "words". The passphrases tested as subjects for the hypothesis came from several different dictionaries varying in different sizes of words and different parts of speech, and were built in a variety of ways, such as in a random order, in a specific order, or as a sentence. The results show that in general, an exact copy of the passphrases resulted in much lower recall accuracy than short passwords after a brief period of time, and only slightly lower recall accuracy than the short passwords after two days. However, when case was not considered important (for example, a word in the passphrase could be entered as 'firetruck' or 'FireTrucK'), the accuracy of recall of passphrases after two days was often slightly better than that of the short passwords. This is promising, because it implies that with a larger dictionary, a passphrase composed of a sequence of words could give a higher entropy than a random sequence of characters with the same recall accuracy. The study also notes that the ability of participants to recall a passphrase was more highly affected by the

number of characters than the number of words, which suggests that a large dictionary of short words could be effective for building random passphrases with a good recall rate [15].

An even more recent paper published in 2015 by Ghazvininejad and Knight, *How to Memorize a Random 60-Bit String*, had an even more involved approach to the passphrase solution [4]. In this paper, passphrases are generated from random 60-bit strings. Similar to the previous paper, the study tested perfect recall accuracy after 2 days. Of the few methods tested, the best results came from the XKCD approach expanded to have a dictionary of size 32,768 ($2^{15}$) rather than size 2,048 ($2^{11}$), and a poetry approach. The poetry approach was considerably more involved, assigning iambic information (focusing on the stresses of syllables in words) to all dictionary words, creating rhyming pairs, and then generating random two-line poems that make a reasonable amount of sense as a poem. Between the two best approaches, the XKCD approach is preferable. The reason for this is that while poetry is somewhat more appealing, it requires the creation of the poems beforehand, which makes it more difficult to adjust in scale, and it just barely did better than the XKCD approach. As a result, while it did ever-so-slightly better, it is much more difficult to implement and scale, which is probably not worth the marginal improvement in practice. This paper also noted that while people reportedly prefer sentence passphrases to random-sequence-of-words passphrases, they actually tend to be worse at remembering sentences. This makes sense when considering how people store ideas: when people remember a sequence of words without any semantic meaning, they focus more closely on the words themselves, whereas when people remember a sentence they store the semantic meaning of the sentence more accurately than the actual content, so when people attempt to recreate a sentence they often match the interpretation of the sentences but do not perfectly replicate the words themselves [4].

### Mock-Implementation of a Password System

The entire implementation for this project, including any required data files, can be found at `https://github.com/mjones05/Passwords`. The implementation in this project is a Python mock-up of a system for generating passphrases and validating credentials. Specifically, the program has a dictionary of common English words, trimmed to 4,300 words in an attempt to make recall easier. Some trimming involved taking out homophones, compound words, and commonly-misspelled or difficult-to-spell words. The approach to building passphrases is a random sequence of words, similar to the approach presented by XKCD [14], but using 5 words instead of 4 words in order to gain higher entropy. The process of entering passwords involves typing out the whole passphrase with spaces, but the input is not case-sensitive, to further aid correct recall without drastically reducing entropy. Passwords are encrypted using SHA256 encryption with salting.

The basic functionality of the program involves reading the word list, a list of SQL reserved words, and a list of preset credentials, and then allowing the user to perform a few commands. In the set of commands, the user can:

- add a new username, which will be associated with a new and randomly-generated salt and password pair.
- remove a username, including data on the salt and password.
- change the password for a username, including the salt. As with all password creation in this program, the salt and password will be randomly-generated.
- 'login' with a username and password. This doesn't actually log in to anything, it simply verifies whether the password is correct for the username (if the username exists).

There are a couple additional features to help demonstrate functionality of a password system of this type. First, even though there is not really an 'admin' or 'non-admin', any error messages

are displayed as 'to the user' or 'to admin', to demonstrate how such a system would minimize information leakage. Second, there is a list of reserved words in SQL, and the list of words used in passwords is disjoint from the list of reserved words. As a result, any time a password is input which is not valid or is incorrect due to a nonexistent username or incorrect hash, the system will also perform a cursory check for SQL keywords and will send an error message to the admin if any SQL keywords are detected. Note that the remove and change commands are actually two commands each, one of which represents a call by an admin and one of which represents a non-admin call, to demonstrate how the commands would behave differently under each of the two conditions. Finally, the system does implement case-insensitive password checking, and works correctly under those assumptions.

# Results and Analysis of the Implementation

## Basic Results

In general, the mock implementation of such a system was successful. At the most basic definition of success, the program is able to randomly generate passphrases and salts, and the program can correctly identify a username/password pair as valid or invalid, with the password being case-insensitive. This includes the process of hashing with salting, which implies that this process could be implemented in a real system on the back-end without storing unencrypted passwords.

The program was also able to successfully report information differently to administrators and non-administrators, to allow discreet detection of a basic SQL injection attack and prevent information leakage. In the realm of SQL, this code is able to detect only simple SQL injections. Note that we only use a very simple pseudo-database, and the entirety of the program is in Python, so no real SQL injection can occur, so complete testing for SQL injection prevention does not make sense. However, this program will detect SQL keywords separated by spaces. It will not detect SQL injections which avoid using spaces, and therefore additional processing would be necessary to catch such cases and prevent real injection in an actual implementation. However, the careful selection of which users see which information was successful. First, SQL keyword detection is reported only to administrators. Second, non-administrators get error messages only after entering all information, and only get error messages as generic as possible. This means that when an incorrect username is entered, a password is still prompted, SQL keyword detection still occurs, and users only get the error message 'Invalid username/password' in all cases. Administrators performing commands get more detailed error messages about where the error occurred.

It is important to note that this is not a complete system, by any means. The only pieces which this program successfully demonstrates entirely is the ability to produce and validate passwords and salts for usernames, and the ability to give commands with different privileges different amounts of information in error messages. To prevent injection, considerably more processing details are needed. To make sure that security is well-maintained, access control would need to be managed with respect to the code in 'passwords.py' and all of the related .txt files. To prevent brute-force guessing, checks would need to be put in place to limit the number of incorrect guesses for a username, as well as some sort of system to prevent repeatedly attempting to create usernames to mine for existing usernames. Finally, it would be better to have some small study with the success of the passwords, specifically with respect to recall accuracy, to confirm that the password system works at similar password systems in other papers.

However, these issues are beyond the scope of this paper. This implementation certainly served as the proof-of-concept it was intended to be. This program demonstrates that the ability to generate and validate passphrases as sequences of words is definitely achievable with a reasonable amount of work.

### Analysis

Unfortunately, no data exists for the recall rates of the passwords with this design, so no such data will be discussed here. Instead, this section will focus on the security of the randomly-generated passwords.

The passwords from this program are reasonably secure. In terms of entropy, the entropy of each word is $\log_2 4300 \approx 12$, so the total entropy of a password is $5 * \log_2 4300 \approx 60$. This means that the entropy of each password is the same as any of the passwords in *How to Memorize a Random 60-Bit String* [4]. This means that this yields nearly 1.5 quintillion password combinations with equal probability, which is about the same number of random 10-character passwords using the set of alphanumeric characters. Even a computer which could operate at 100 billion password guesses per second would require over 5 months to break a single password of this form. On top of that, since incrementing from one password guess to the next requires changing multiple bytes of data, an incremental search on this password form would likely be less effective than an incremental search of a random 10-character password, scaling the time required to break a password even more.

Other difficulties with cracking passwords of this form arise when the cracker doesn't have the word set for the passwords. If it were possible to restrict access to the word list, even if the hashes were released with the salts, it's difficult to build an efficient cracker without a word list. If the attacker only uses some of the words, which they could potentially learn by creating usernames or cracking some passwords by another method, it becomes difficult for them to be sure that they can get a large number of passwords without adding some words to their search. However, doing this would involve using another word list which is possibly much larger. For example, a list of English words at `https://github.com/dwyl/english-words/` contains nearly 500,000 English "words". Most of these words are nonsense, such as '&c' or 'ragamuffinism', but it is still useful for making the point that the word list used in the implementation is only a small subset of all English words, and using a word list which is too large of a superset of the password's word set can be incredibly detrimental to the efficiency of a password cracker. While it is difficult to prove that this approach has a better recall rate than a random sequences of alphanumeric characters, studies suggest that it is likely that it does [4, 15] and it is very strong against password crackers, even if the attacker has a copy of the word list.

## Discussion

The key question this paper sets out to answer is whether or not a password system involving words as building-blocks for passphrases is achievable, and whether or not such a system is worth the effort to build. While this is a very open-ended question, it is important to consider all of the points on both sides of the argument. The three main components of the argument are the amount of work required to build such a password system, the effect of the system on security, and the effect of the system on the users' ability to correctly remember and input passwords.

Overall, the amount of work to build this system is not unmanageable. The design of the system to parse passwords, remove any items involved in a malicious attack, and verify whether or not a username/password is valid is only marginally more difficult than a similar system for any other password format. The ability to detect basic SQL injection is slightly easier for this approach, because many of the tools for parsing the passwords can be applied directly for this purpose, most of the additional work is simply building a list of words that are reserved in SQL. However, in terms of a complete protection from injection, this system would need the same checks that any other password system would need. As a result, the implementation is only slightly more difficult than a general password system.

In the mock implementation, the bulk of the work involved with this creating this password system was actually heavily focused on the creation of the word list. The difficulties with the word lists are numerous, including:

- Dealing with homophones. A user is less likely to get the password correct if they can't memorize the words phonetically. If a possible password is 'bear bare bar flew flu', a user with that password will have considerably more difficulty remembering that password correctly than a password such as "orange purple grass yellow brown", which is much easier to memorize.

- Dealing with compound words. An example of this would be the word 'barefoot' versus 'bare foot', which sound very similar and are therefore difficult to differentiate when remembering a passphrase.

- Dealing with difficult-to-spell words. For example, while the word 'onomatopoeia' is a word with a very unique pronunciation, it would probably be unwise to use this word in a passphrase because the user would probably either frequently misspell the word or have to look up the spelling every time they go to enter their password.

While some of these issues are able to be dealt with relatively strictly, others are more difficult for a computer to identify, and some are simply very subjective. While it is possible to use heuristics, such as removing any word that is a concatenation of two other words, the subjective issues are more difficult to solve. For example, to only use easy-to-spell words, a computer could only use words with length less than 9. However, this is not perfect, because it will keep some words which may be tricky to spell (such as ecstasy, achieve, or bough) and will omit words which are generally easy to spell (such as unhelpful or unimportant). Even the issues of compound words and homophones can cause issues for computers, such as the edge case involving 'ful' and 'full'. For example, 'wonderful' and 'wonder full' sound very similar, but looking for compound words using the earlier approach would miss this case. With respect to the mock implementation, even after hours of combing through the word list and trimming using heuristics, possible edits such as removing 'wonderful' can still be identified. It is also difficult to define a single word list and use that for every instance of this password system, since standards such as the maximum difficulty of spelling words or the tolerance for words such as 'violence' or 'stupid' can vary depending on the expected users. It is difficult to find the balance between a set of words such that is easy for people to remember passphrases while keeping the list large enough to give the passwords sufficient entropy. As a result, word lists are the largest issue for this approach, since

it is difficult to standardize word lists across settings and therefore requires a large number of man-hours to design the word lists. This issue also creates an issue across languages, since a word list would need to be created in each language.

In terms of the security of this system, the entropy and randomness of the system is quite sufficient for protecting passwords from all but the fastest supercomputers, as shown in the Basic Results section. The design is certainly sufficient for the average case of required security. In terms of the ability to memorize the passwords, no study was conducted using the mock implementation. However, based on the results from [4, 15], it is likely that these passwords would have a much higher rate of recall than a string of random characters with the same entropy. Before this system could be applied in a real setting, it would also be wise to conduct studies with respect to actual users to determine if users would settle on similar passwords. Specifically, would some words be preferable to others? If this were the case, people would be more likely to change passwords with other words, which would reduce the entropy of such passwords. While this could be prevented by not allowing users to manually call for a password change or limiting the number of times a user could call for a password change, such restrictions would be unwise without information about how well the average passphrase can be recalled and how satisfied a user is with the passphrase.

This leaves the question of whether or not this is an ideal approach to a password system very open-ended. On a case-by-case basis, the system designers should look carefully at the requirements of passwords for their system. If there is little to no sensitive data or negative ramifications of unauthorized access, then password strength may be unimportant, in which case a standard user-created password with certain restriction on the character set and minimum length would be ideal. However, if password security for a user of the system is a significant factor, this may be a better approach to passwords than the current common practices. At the moment, such an approach would likely be implemented only by the largest entities where this is a significant issue, due to the resources required to define effective word lists. However, an excellent next step would be to design a couple large word lists, with different levels of spelling difficulty and available words by other metrics, so that the bulk of the work for designing a passphrase system is already handled and it would be generally easier for anybody to implement this system. Until such word lists exist, it is probably unlikely that this approach would see widespread use in practice.

## Conclusions

With respect to the specific passphrase method proposed in this paper, the results show that such methods are reasonable to implement, they may require a large amount of design work. The passwords created using these approaches have sufficient entropy to show they are secure, and results from other studies suggest that they are more memorable than randomly-generated passwords of the same entropy [4, 15]. In order for this approach to be widely accepted in practice, a considerable amount of work may have to go into designing sufficiently efficient, effective, and large word lists which are publicly available. The implementation here is sufficient as a proof-of-concept, but is unable to solve such issues which would occur in a realistic setting. At the very least, the results of this paper act to show that the current approaches to password systems are not generally effective against a smart attacker, and that the tradeoff these systems make between the ability to remember passwords and the security of those passwords is unnecessary and unwise. Therefore, a passphrase system with computer-generated passphrases is often an attractive and practical idea in situations where password security is a significant factor.

## References


[1] Polybius. (1962). *The histories of Polybius. Bloomington: Indiana University Press*

[2] Goodin, D. (2012, December 9). 25-GPU cluster cracks every standard Windows password in. Retrieved from https://arstechnica.com/information-technology/2012/12/25-gpu-cluster-cracks-every-standard-windows-password-in-6-hours/

[3] Gao, X., Yang, Y., C., Mitropoulos, C., Lindqvist, J., & Oulasvirta, A. (2018). Forgetting of Passwords: Ecological Theory and Data. USENIX Security Symposium, 221-238. Retrieved December 12, 2018.

[4] Ghazvininejad, M., & Knight, K. (2015). How to Memorize a Random 60-Bit String. HLT NAACL. Retrieved from https://www.isi.edu/natural-language/mt/memorize-random-60.pdf.

[5] Shannon, C.E. Prediction and Entropy of Printed English.

[6] Yazdi, S. H. (n.d.). Analyzing Password Strength & Efficient Password Cracking. Retrieved December 12, 2018, from http://purl.flvc.org/fsu/fd/FSU_migr_etd-3737

[7] Shay, R., Komanduri, S., Durity, A. L., Huh, P., Mazurek, M. L., Segreti, S. M., ... Cranor, L. F. (2016). Designing Password Policies for Strength and Usability. TISSEC, 18(4). doi:10.1145/2891411

[8] Wang, C., Jan, S. T., Hu, H., Bossart, D., & Wang, G. (2018). The Next Domino to Fall: Empirical Analysis of User Passwords across Online Services. ACM CODASPY, 196-203. doi:10.1145/3176258.3176332

[9] Wash, R., Rader, E., Berman, R., & Wellmer, Z. (2016). Understanding Password Choices: How Frequently Entered Passwords Are Re-used across Websites. SOUPS, 175-188. Retrieved December 12, 2018, from https://www.usenix.org/system/files/conference/soups2016/soups2016-paper-wash.pdf.

[10] Ur, B., Noma, F., Segreti, S. M., Shay, R., Bauer, L., Cristin, N., & Cranor, L. F. (2015). "I Added '!' at the End to Make It Secure": Observing Password Creation in the lab. SOUPS, 123-140. Retrieved December 12, 2018, from https://www.usenix.org/system/files/conference/soups2015/soups15-paper-ur.pdf.

[11] Zhang, Y., Monrose, F., & Reiter, M. K. (2010). The Security of Modern Password Expiration: An Algorithmic Framework and Empirical Analysis. CCS, 176-186. doi:10.1145/1866307.1866328

[12] Bonneau, J. (2012). The Science of Guessing: Analyzing an Anonymized Corpus of 70 Million Passwords. 2012 IEEE Symposium on Security and Privacy. doi:10.1109/sp.2012.49

[13] Kurzban, S. A. (1985). Easily Remembered Passphrases–A Better Approach. ACM SIGSAC Review - Resources: Part II, 3(2-4), fall/winter 1985, 10-21. doi:10.1145/1058406.1058408

[14] Munroe, R. (n.d.). Xkcd - Password Strength [Cartoon].

[15] Shay, R., Kelley, P. G., Komanduri, S., Mazurek, M. L., Ur, B., Vidas, T., . . . Cranor, L. F. (2012). Correct horse battery staple: Exploring the usability of system-assigned passphrases. Symposium On Usable Privacy and Security. doi:10.1145/2335356.2335366