A Comparison of Three Random Password Generators

Michael D. Leonhard
College of Engineering, University of Illinois at Chicago, Chicago, Illinois, U.S.A.
uic@tamale.net

Abstract

This paper compares three random password generation algorithms, describing and analyzing each. It also reports the results of a small study testing the quality of the passwords generated by the algorithms. Qualities discussed include security, memorability, and user affinity. Suggestions are made for improving the experiment and the individual algorithms.

1 Introduction

Passwords are the most common user authentication method in use by multi-user computer systems. Some systems allow each user to choose her own password. Other systems create a random password for the user. This paper compares three algorithms for creating passwords.

Random passwords are commonly found in high-security systems. The military makes extensive use of random passwords [1]. Generally, random passwords are used for one-time authentication and for applications where the user is expected to memorize the password and not write it down.

Random passwords have several benefits over user-chosen passwords. The generator algorithm guarantees that the password contains a specific amount of entropy. This means that the password is chosen from a set of potential passwords that is large. An attacker must search through many potential passwords to break the authentication mechanism. On the other hand, when a user selects a password, there is no guarantee that the password comes from such a large set. Users often choose simple passwords that contain only a word and a number. The set of such passwords is very small. This is why user-chosen passwords are much easier to attack than random passwords.

Some researchers have suggested instructing users to create mnemonic phrase-based passwords. The premise is that such passwords will not appear in password cracking dictionaries and will thus be more resistant to attack. But Kuo, Romanosky, and Cranor showed that one can build a dictionary for such passwords [2].

A user will often use the same password for multiple applications [3]. When an attacker compromises one application, he learns the user's password to the other applications. By using the same password on multiple accounts, the user is setting up a fragile security where a single breach leads to total loss of security. Random passwords increase security by forcing the user to use a unique password for the application. This is limited by the fact that the user may adopt this application's random password for use on various other accounts.

Both of these problems with user-chosen passwords can be avoided if the user behaves as instructed. The user can select a good password from a large password set. She can use a unique password for each application. But it is unrealistic to expect perfect compliance from users. In many cases, the user does not even benefit from the presence of an account and password [3]. When the user is not motivated to follow password policies, the application can force compliance by assigning a random password [4].

Random passwords are more difficult to remember than user-chosen passwords. When given the opportunity, a user will choose a password that has meaning to her [3]. She will have mental connections to the password and can remember it. An assigned password has no intrinsic meaning to the user. Many successful memorization strategies consist of finding meaning in the random password and building mental connections. Unmotivated users loathe expending such effort. The difficulty of remembering a random password may drive the user to

write down her password or simply stop using the application. Thus it is imperative that random password generators use the best known algorithm.

The purpose of this study is presented below, in Section 2. Related work is outlined in Section 3. Section 4 describes the experimental procedure. I then describe and analyze the password generators, in Section 5. The results of the experiment are in Section 6. Conclusions drawn from the experiment appear in Section 7. Future work is described in Section 8.

2 Purpose

This study considers three algorithms, named AlphaNum, Diceware, and Pronounce3. They were chosen because they are all very easy to analyze and they come from different classes of algorithms. The AlphaNum algorithm constructs sequences of random characters. Diceware creates passphrases. Pronounce3 constructs strings of syllables.

The purpose of this study is to find out which password generator produces the best quality passwords. I consider the following qualities of passwords:

1. Security: the amount of entropy in each password
2. Memorability: how easily a normal person can remember the password
3. Affinity: how much the user likes the password

There are various other characteristics of passwords that are not considered: password length and language.

3 Related Work

A similar study was performed by Bunnel, et. al. [5]. They compared user-generated passwords, randomly generated passwords, question-answer pairs, and word associations. Their participants correctly recalled 77% of user-generated passwords and 70% of randomly generated passwords. Their random password algorithm was very simple. It concatenated a three-letter word, a numeral from 1 to 9, and a four-letter word. Although the security of the algorithm is unsatisfactory, their study produced valuable experimental data. Their experiment served as a model for my study, presented here.

The US Department of Defense published guidelines for password management [1]. They present a technique for analyzing the security of passwords. I employ that technique in this study. They also suggest algorithms that are very similar to the AlphaNum and Diceware algorithms that I present here.

4 Procedure

The experiment consists of administering two questionnaires. The first contains a randomly generated password and tasks intended to help the participant to memorize it. The second questionnaire, given two weeks later, asks the participant to recall the password. The participants are undergraduate and graduate students taking a class on network security. The participants are likely to have a high understanding of security concepts and good password practices. To offset this, the questionnaire instructs the participants to treat the passwords as they would any other password.

For the first questionnaire, I ran each algorithm's Python script to obtain 20 random passwords. This yielded 60 random passwords altogether. I mixed up the order of the passwords. An AlphaNum password was first, then a Diceware password, then a Pronounce3, then another AlphaNum, and so on. I printed a questionnaire for each password. I handed out the questionnaires to participants by row, so people sitting next to each other would not receive the same type of password. Also, I distributed equal numbers of passwords of each type.

The first questionnaire contains instructions and a mockup webpage interface for a fictional website called Joe Maxwell Internet Auctions. The participant is to role-play as a user of the website. Every view of the website contains the same logo and title.

The first "page" thanks the user for registering and displays her randomly generated password. Three subsequent "login screens" request the user to write her password in the password box and log in. If the user were to complete the questionnaire in a few moments, it is unlikely that she will remember the password later. Based on the assumption that retention is enhanced by lengthening the period for memorizing, I added meaningless time-consuming tasks between the login screens. The participants completed the first questionnaire in about 5 minutes.

The second questionnaire was administered two weeks after the first. It instructs the participant to role-play logging into the website again. Three login screens are given, which are identical to those presented in the first questionnaire. Instructions ask the participant to try to remember her password and write it in the first login screen. Then, if she is uncertain of the password's correctness, she is to write other passwords that may be correct in the second and third login screens. The questionnaire then asks some multiple choice questions. Next, there are two open-ended questions and space to write in responses. Finally, there is a space for the participant to write her email address if she wishes to receive a summary of the study results.

The second questionnaire sheets were all identical. The first questionnaire sheets each contained a password generated by an algorithm presented below.

5 Algorithms

There are three algorithms. For each algorithm, I describe the technique used to generate passwords. This is followed by an analysis of the security of the passwords. I present a list of twenty passwords generated by that algorithm. These are the passwords used in the experiment. Finally, I present a Python script implementing the algorithm.

5.1 AlphaNum Algorithm

This is the simplest generator. It creates a random password that is 6 characters long and may contain upper-case letters, lower-case letters, and numbers. The size of the alphabet is 26 + 26 + 10 = 62. The algorithm chooses from this alphabet six times. The resulting password is the result of these six choices. There are 62 possibilities for the first character, 62 for the second, and so on. So the number of possible passwords is:

$$62 \times 62 \times 62 \times 62 \times 62 \times 62 = 62^6 = 5.68 \times 10^{10} = 2^{35.7}$$

That is the size of the password set. Mathematically, let P_A be the set of all possible passwords generated by this algorithm. The size of the set is called the cardinality of P_A, denoted $|P_A|$. Because $|P_A| = 2^{35.7}$, we say that any password, p_A, chosen randomly from P_A, has 35.7 bits of entropy. We can use this measure of entropy to compare the strength of various algorithms.

The purpose of this generator is to make passwords that are very short, yet contain enough entropy.

```
1. 6m4CYM
2. IFvA8L
3. dVysgZ
4. a1LCLQ
5. EDaL8p
6. u1pbqY
7. DbKrRZ
8. ED0uPw
9. tIG6QL
10. R7oBwn
11. YsM8Ht
12. YpD1fD
13. B1bWcn
14. XKu6ad
15. 1EKmOQ
16. qXvyK1
17. g9J2Lx
18. O3Itgc
19. A1ZXnN
20. OKg8qc
```

Figure 1. Output of alphanum.py

The program in Figure 2 is the implementation of AlphaNum in Python. Figure 1 is the output of the program running on Python 2.3.4 on Linux.

```
#!python
#
# alphanum.py - generates random alphanumeric passwords
# Copyright (C) 2006 Michael Leonhard
import random

LowerCase = "abcdefghijklmnopqrstuvwxyz"
UpperCase = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
Numbers = "0123456789"

Set = LowerCase + UpperCase + Numbers

def genPwd():
    Password = ""
    for N in range(0,6):
        Password = Password + random.choice(Set)
    return Password

for N in range(1,21):
    print "%d. %s" % (N, genPwd())
```

Figure 2. Source code for AlphaNum generator, alphanum.py

5.2 Diceware Generator

This algorithm produces random lists of words. It uses the idea that memorization requires one to form mental connections to the information being memorized. Every person who knows the meaning of a word has some kind of mental connection to it. Thus by forming passwords with words, the person can take advantage of the existing mental connections to make memorization easier. That is the theory.

This kind of password is called a 'passphrase' by the DoD [1]. Reinhold provides a list of 7,776 common words on his website [6]. He explains how to select passphrases using common six-sided dice.

The program in Figure 4 is my implementation of Diceware in Python. The program uses Reinhold's English wordlist from his website [7]. Before running the program, one must process the wordlist file with this unix command:

```
cat diceware.wordlist.asc |grep -E
 "^[123456].*" |cut -f 2 > words
```

This command extracts the lines containing numeric labels and then strips those labels, leaving only the desired words.

```
1. kraut gwen nagoya
2. hirsch ay qed
3. voss terre snub
4. plaid hey benz
5. scope movie gouda
6. isis uptake rca
7. mph scm ranch
8. bryce aspire clone
9. doze stuff salve
10. doe slim dodo
11. lv spiky coat
12. fusty leper avon
13. portia toe trunk
14. lares ave ghent
15. fed saga greet
16. pion ride bile
17. cyrus comet 99
18. those rascal wall
19. dixie frost cv
20. ample acidic leery
```

Figure 3. Output of diceware.py

The algorithm independently chooses three words from the word list. Thus

$$|P_D| = 7776^3 = 4.70\times10^{11} = 2^{38.8}$$

This algorithm, Diceware, produces passwords with 38.8 bits of entropy. It is a little bit stronger than AlphaNum, which has 35.7 bits.

```
#!python
#
# diceware.py - generates passphrases from the diceware.org wordlist
# Copyright (C) 2006 Michael Leonhard
import random, string

Words = map(string.strip, file("words","rU").readlines())

def genPwd():
    Password = None
    for N in [1,2,3]:
        NewWord = random.choice(Words)

        if None == Password:
            Password = NewWord
        else:
            Password = Password + " " + NewWord
    return Password

for N in range(1,21):
    print "%d. %s" % (N, genPwd())
```

Figure 4. Source code for Diceware Generator, diceware.py

Figure 3 is the output of the program running on Python 2.3.4 on Linux. As you can see, some of the words are rather obscure. Passwords may contain words that users do not know. For example, the author is unaware of the meanings of 'portia', 'lares', and 'ghent'. For future work, less common words may be removed from the wordlist. Reinhold's technique utilizes six-sided dice and requires 7776 words. But the program in Figure 4 is not limited in this way. It can benefit from a slightly shorter word list that contains only words in common use today.

5.3 Pronounce3 Generator

This algorithm produces passwords that are pronounceable in English. The objective of this algorithm is to utilize the speech facilities of the user's mind to assist in remembering the password.

Ganesan and Davies describe a major flaw in pronounceable password generators [8]. The generators choose syllables based on their frequency in English writing, using complex rules to achieve pronounceability. The result is that some passwords are more likely to be chosen than others. Ganesan and Davies show how this lack of uniform probability ruins the security of the algorithms.

The Pronounce3 algorithm does not have the flaw described by Ganesan and Davies. It takes a simple approach to password construction resulting in uniform entropy for all passwords in the password space. We can easily analyze the security of the algorithm.

The Pronounce3 generator composes passwords of consonant and vowel elements. There are five vowels:

a, e, i, o, u

```
1. adustphelo
2. ahzuphoste
3. zuenacha
4. vubagese
5. zuwelopu
6. agrofuxa
7. fustuwchoi
8. ezvedoxe
9. yechnopee
10. ulciyolu
11. epchigaxu
12. aphoduwzu
13. abdaumso
14. cudawigo
15. voyuvephu
16. inunwizi
17. acistjalu
18. urcezfae
19. phoupodphu
20. owuphiwpi
```

Figure 5. Output of pronounce3.py

There are twenty two consonants:

b, c, ch, d, f, g, h, j, k, l, m, n, p, ph, r, s, st, v, w, x, y, z

To ensure consistency with English spelling, we restrict password composition with two rules:

1. No password may begin or end with two consonants.
2. The password may not contain three consecutive consonants or three consecutive vowels.

Given a certain number of vowels and consonants, there are various orderings that satisfy the two restrictions. The algorithm represents an ordering with a template string. A template is a string of 'a' and 'b' symbols, where 'a' represents a vowel and 'b', a consonant. The set of templates with v vowels and c consonants is denoted $T_{v,c}$. The set of templates using 4 vowels and 4 consonants contains thirteen elements:

$T_{4,4}$ = {aabbabba, ababbba, ababbaba, abbaabba, abbababa, abbabbaa, baababba,
baabbaba, babaabba, babababa, bababbaa, babbaaba, babbabaa}

Given sets of templates, vowels, and consonants, password generation begins by randomly choosing one of the templates. The algorithm then iterates through the template. When an 'a' is encountered, it randomly chooses a vowel and appends it to the password. Each vowel is equally likely to be chosen. Similarly, for a 'b', it appends a random consonant.

Let us denote the set of all passwords generated by the algorithm as $P_{v,c}$ where v is the number of vowels and c is the number of consonants. Since $T_{v,c}$ is the set of valid templates that contain v vowels and c consonants, it should be plain that

$$|P_{v,c}| = |T_{v,c}| \times 5^v \times 22^c$$

For this study, I use the Pronounce3 algorithm to generate passwords containing 4 vowels and 4 consonants. Figure 6 is a program in Python that implements this. Figure 5 is the output of the program running on Python 2.3.4 on Linux. The algorithm chooses passwords from the set $P_{4,4}$.

$$|P_{4,4}| = 13 \times 5^4 \times 22^4 = 13 \times 625 \times 234256 = 1.90\times10^9 = 2^{30.8}$$

The algorithm's 30.8 bits of entropy are less than AlphaNum's 35.7 bits and Diceware's 38.8 bits. I considered several ways to increase the entropy of this algorithm. One way is to introduce more templates. This requires different numbers of vowels and consonants. Table 1 lists the eighteen non-empty password sets whose passwords have length eight or less. Note that this length is the number of vowel and consonant elements. Some elements, such as 'ch', contain two characters. Passwords containing such elements are longer than eight characters.

From the table, we can see that $P_{4,4}$ is the largest set. As shown previously, using only $P_{4,4}$ yields passwords with 30.8 bits of entropy. Consider modifying the algorithm to choose passwords from $P_{4,4} \cup P_{3,5}$.

$$|P_{4,4}| + |P_{3,5}| = 1.90\times10^9 + 6.44\times10^8 = 2.54\times10^9 = 2^{31.2}$$

By adding $P_{3,5}$, we gain a negligible 0.4 bits of entropy. Can we do better if we include all of the valid password sets? Let's see:

$$|P_{1,0}| + |P_{1,1}| + |P_{2,0}| + \ldots + |P_{5,3}| + |P_{6,2}| = 3.16\times10^9 = 2^{31.5}$$

This is hardly any better than using only $P_{4,4}$. In fact, by using all of the sets, we gain only 0.76 bits of entropy. Clearly, to achieve higher entropy, the algorithm must allow some templates that contain nine elements. That is an area for future study.

v	c	$\lvert P_{v,c} \rvert$	$\lvert T_{v,c} \rvert$	$T_{v,c}$
1	0	5.00×10^{0}	$T_{1,0} = 1$	a
1	1	1.10×10^{2}	$T_{1,1} = 1$	ba
2	0	2.50×10^{1}	$T_{2,0} = 1$	aa
2	1	1.10×10^{3}	$T_{2,1} = 2$	aba, baa
2	2	2.42×10^{4}	$T_{2,2} = 2$	abba, baba
2	3	2.66×10^{5}	$T_{2,3} = 1$	babba
3	1	5.50×10^{3}	$T_{3,1} = 2$	aaba, abaa
3	2	3.03×10^{5}	$T_{3,2} = 5$	aabba, ababa, abbaa, baaba, babaa
3	3	6.66×10^{6}	$T_{3,3} = 5$	ababba, abbaba, baabba, bababa, babbaa
3	4	8.78×10^{7}	$T_{3,4} = 3$	abbabba, babab ba, babbaba
3	5	6.44×10^{8}	$T_{3,5} = 1$	babbabba
4	1	1.38×10^{4}	$T_{4,1} = 1$	aabaa
4	2	1.51×10^{6}	$T_{4,2} = 5$	aababa, aabbaa, abaaba, ababaa, baabaa
4	3	7.32×10^{7}	$T_{4,3} = 11$	aababba, aabbaba, abaabba, abababa, ababbaa, abbaaba, abbabaa, baababa, baabbaa, babaaba, bababaa
4	4	1.90×10^{9}	$T_{4,4} = 13$	aabbabba, abababba, ababbaba, abbaabba, abbababa, abbabbaa, baababba, baabbaba, babaabba, babababa, bababbaa, babbaaba, babbabaa
5	2	4.54×10^{6}	$T_{5,2} = 3$	aabaaba, aababaa, abaabaa
5	3	4.33×10^{8}	$T_{5,3} = 13$	aabaabba, aabababa, aababbaa, aabbaaba, aabbabaa, abaababa, abaabbaa, ababaaba, abababaa, abbaabaa, baabaaba, baababaa, babaabaa
6	2	7.56×10^{6}	$T_{6,2} = 1$	aabaabaa

Table 1. All eighteen non-empty password sets and their properties.

Another area to investigate is the addition of capital letters. By allowing the first letter to be either upper-case or lower-case, we gain one bit of entropy. Various other capitalization schemes deserve investigation, too. Another promising modification is the addition of symbols such as the hyphen, period, asterisk, etc.

```
#!python
#
# pronounce3.py - a random pronounceable password generator
# Copyright (C) 2006 Michael Leonhard
import random, string, sys

# recursively builds a list of all possible strings containing NVowels 'a'
# and NConsonants 'b'.
def makeTemplates(PartialString, NVowels, NConsonants):
    Result = []
    if NVowels > 0:
        Result.extend(makeTemplates(PartialString+"a", NVowels - 1, NConsonants))
    if NConsonants > 0:
        Result.extend(makeTemplates(PartialString+"b", NVowels, NConsonants - 1))
    if 0 == NVowels and 0 == NConsonants:
        Result.append(PartialString)
    return Result

# checks if the string would be a legal template for building a password
def templateFilter(Template):
    A = ""
    B = ""
    for C in Template:
        # disallow two leading consonants
        if (A,B,C)==("","b","b"):
            return False
```

```
        # disallow three consecutive consonants
        if (A,B,C)==("b","b","b"):
            return False
        # disallow three consecutive vowels
        if (A,B,C)==("a","a","a"):
            return False
        A = B
        B = C

    # disallow two final consonants
    if (B,C)==("b","b"):
        return False
    return True

Templates = filter(templateFilter, makeTemplates("", 4, 4))
Vowels = ["a","e","i","o","u"]
Consonants = ["b","c","ch","d","f","g","h","j","k","l","m","n","p","ph" \
,"r","s","st","v","w","x","y","z"]

def genPwd():
    Password = ""
    Template = random.choice(Templates)
    for C in Template:
        if "a"==C:
            Password = Password + random.choice(Vowels)
        elif "b"==C:
            Password = Password + random.choice(Consonants)
        else:
            raise "ERROR"
    return Password

for N in range(1,21):
    print "%d. %s" % (N, genPwd())
```

Figure 6. Source code for Pronounce3 Generator, pronounce3.py

6 Results

The experiment used the passwords in figures 1, 3, and 5. Twenty nine people participated in the first part of the experiment, receiving a password on the first questionnaire. Nineteen of those people completed the second part of the experiment, properly filling out the second questionnaire. Table 2 lists the distribution of passwords from various algorithms to the students and their recollection rate.

	AlphaNum	Diceware	Pronounce3
Total Participants	6	7	6
Recalled Password	1	2	1

Table 2. Participants completing both questionnaires and recalling their passwords.

No participant wrote an incorrect password in the first login box and subsequently wrote a correct password in the second or third boxes. If the first response was incorrect, so were the others. Some participants recalled their passwords but were mistaken in one letter. Others left out a letter. Table 3 shows data from the questionnaires of the nineteen participants.

Assigned Password	First Response	Levenshtein Distance	Confidence of correct response	Trick Question ('yes' = correct)	Wrote down password	Affinity	Attempted Equation	Solved Equation
EDaL8p	eDALp8	4	don't know	yes	no	don't like it	yes	no
ED0uPw	DP0sp1	5	no	no	no	hate it	no	no
tIG6QL	tI6QL	1	probably	yes	yes	love it	yes	no
R7oBwn	R7oBwm	1	yes	yes	no	ok	yes	yes
YsM8Ht	YsM8Ht	0	probably	yes	yes	ok	yes	yes
YpD1fD		6	no	yes	no	don't like it	yes	yes
voss terre snub	bryan bruce …	13	probably not	yes	no	ok	yes	no
plaid hey benz		14	no	yes	no	ok	yes	no
isis uptake rea	hs309	14	no	no	no	don't like it	yes	no
bryce aspire clone	alice Spruce	11	probably not	no	no	don't like it	no	no
doe slim dodo	doe slim dodo	0	yes	yes	no	like it	yes	yes
lv spiky coat	lv spiky coat	0	yes	yes	no	don't like it	yes	yes
fusty leper avon	don't remember	13	no	no	no	ok	yes	yes
agrofuxa	agrofuxa	0	yes	yes	yes	ok	yes	yes
fustuwchoi	qhwch3	7	no	no	no		no	no
ezvedoxe	~doxe	4	no	yes	no	hate it	yes	no
yechnopee	yecknopee	1	probably	yes	no	ok	yes	no
ulciyolu	bayou	5	probably not	no	no	like it	yes	yes
epchigaxu	cguet---	9	no	yes	no	ok	yes	yes

Table 3. Detailed experiment results.

The Levenshtein Distance is the number of edits required to transform their first response into the correct password. It represents how close the user's response was to the correct response. See Figure 7.

The trick question was "Did you write your password on the questionnaires?" The answer should always be 'yes.' The various 'no' responses indicate that some participants did not understand the question, possibly due to insufficient English comprehension.

The Affinity column of Table 3 holds the participants' responses to the question "How do you like your password?" After converting the responses to numerical values, we can compare the responses for the various algorithms. Here is the coding: "hate it" = 0, "don't like it" = 1, "ok" = 2, "like it" = 3, "love it" = 4. Table 4 lists the results of this analysis. The numbers indicate that participants liked the passwords from the Pronounce3 algorithm a little bit more than the other algorithms. Because of the small sample size, this difference is probably within the margin of error.

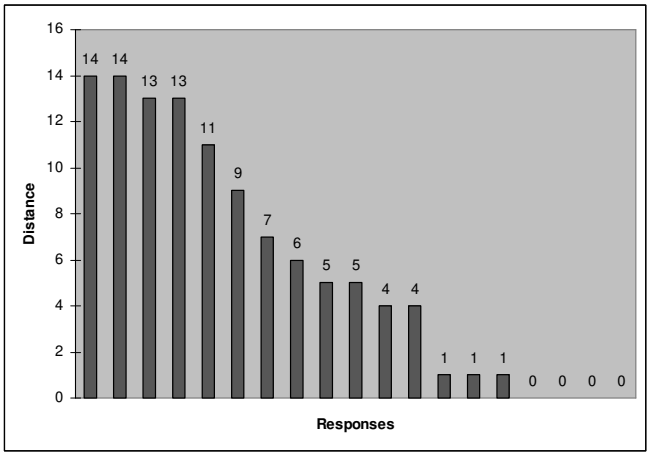

Figure 7. Levenshtein distance of recalled passwords to assigned passwords

	Mean
All Algorithms	1.73
AlphaNum	1.67
Diceware	1.71
Pronounce3	1.83

Table 4. Averages of responses to the question "How do you like your password?"

Responses to the open-ended questions at the end of the second questionnaire were enlightening. Four participants reported using rote memorization. One participant remarked, "I tried to recollect it often (of course, not that frequently)."

Six participants reported using mnemonic techniques to associate meaning with different portions of their passwords. One wrote, "It was very hard to remember, because there were no meaningful words in them that could be remembered."

Four participants indicated that repeated use would have helped them to remember their passwords. One participant wrote, "I don't remember anything well. Only repetition over many days will I remember it."

7 Conclusion

It is unfortunate that only nineteen participants completed both questionnaires. This leaves a sample size of six or seven for each algorithm. This small sample size limits the value of the experimental results. The experiment fails to show significant differences in memorability and affinity among the algorithms. Considering only security, the Pronounce3 algorithm falls behind AlphaNum and Diceware.

8 Future Work

The open-ended question responses direct us to ways we can improve the algorithms. It might be helpful to provide mnemonic aids for AlphaNum passwords. The Diceware algorithm may be improved by removing obscure words from the wordlist. The Pronounce3 algorithm could benefit from the addition of capital letters and punctuation.

Participants' performance also elucidates some areas for improvement. Four participants recalled their passwords perfectly. These show a Levenshtein Distance of 0. Additionally, three participants made only one mistake in their passwords. They have a distance of 1. Perhaps the algorithms may be improved to prevent these minor faults in recollection? For example, one participant incorrectly remembered 'yechnopee' as 'yecknopee'. The person may have memorized the 'ech' sound as 'eck', resulting in an error. Removing the 'ch' consonant element from the Pronounce3 algorithm may be an improvement. Similarly, AlphaNum may be improved by eliminating easily confused pairs such as 'n' and 'm'.

A future experiment should employ a real website in regular use by the participants. A class website would be suitable. The participants would log into the website regularly to download homework assignments and study aids. Each person would use her assigned password regularly. The website could record events such as

successful logins, failed logins, password reminders, etc. This information could form the basis of a better comparison of the password generation algorithms.

References

[1] US Dept. of Defense. *Password Management Guideline*, CSC-STD-002-85, 1985.

[2] Cynthia Kuo, Sasha Romanosky, and Lorrie Faith Cranor. Human Selection of Mnemonic Phrase-based Passwords. *Symposium On Usable Privacy and Security (SOUPS) 2006*, Pittsburgh, Pennsylvania, USA, July 2006.

[3] S. Gaw and Edward W. Felten. Password Management Strategies for Online Accounts. *Symposium On Usable Privacy and Security (SOUPS) 2006*, Pittsburgh, Pennsylvania, USA, July 2006.

[4] J. Yan, A. Blackwell, R. Anderson, and A. Grant. Password Memorability and Security: Empirical Results. *IEEE Security and Privacy*, 2(5), pp. 25—31, 2004.

[5] J. Bunnel, J. Podd, R. Henderson, R. Napier, and J. Kennedy-Moffat. Cognitive, associative and conventional passwords: Recall and guessing rates. *Computers and Security*, Vol. 16, No. 7, pp. 645—657, 1997.

[6] Arnold G. Reinhold. The Diceware Passphrase Homepage, http://www.diceware.com/, accessed 14 October, 2006.

[7] Arnold G. Reinhold. Diceware English Wordlist, http://world.std.com/~reinhold/diceware.wordlist.asc, accessed 14 October, 2006.

[8] R. Ganesan and C. Davies. A New Attack on Random Pronounceable Password Generators. Proceedings of the 17th NIST-NCSC National Computer Security Conference, 1994. pp. 184—197.

Password Memorability Study Questionnaire #1
October 16, 2006
Michael Leonhard

Thank you for participating in this study of password generators. This study compares the quality of passwords generated by various algorithms. You will act as a user of a website. The website generates a random password for you. You will memorize this password by writing it several times. After two weeks, on October 30, you will need to remember the password and log into the website. Please treat this password as you would any normal password of yours. Your participation is greatly appreciated.

Please write your name: _____

Please pretend that you have registered on a website called Joe Maxwell Internet Auctions.

Joe Maxwell Internet Auctions

Thank you for registering. Your password is: **6m4CYM**

To help you memorize your password, please write it in the login box below.

Joe Maxwell Internet Auctions

Login

Password: [ ]

[Login]

Please turn over this page and continue.

Please take a moment and count from 1 through 42 in your mind. Then login again:

Joe Maxwell
Internet Auctions

Login

Password: [ ]

[Login]

Now please solve the following set of equations for y:

$2x = 102 - 2y$
$x = 2y + 42$

Now login again:

Joe Maxwell
Internet Auctions

Login

Password: [ ]

[Login]

That is all. Please return this paper to Michael. The second half of this study will be on Monday, October 30, 2006. Thanks for participating!

Password Memorability Study Questionnaire #2
October 30, 2006
Michael Leonhard

Thank you for participating in my study of password generators! Two weeks ago, you received a sheet like this one. Using that sheet, you registered at Joe Maxwell Internet Auctions, received a password, and practiced logging in. This sheet is the second part of the study. If you choose to participate in this part of the study, please try to remember your password and log in again. If you do not wish to participate, please leave the sheet blank. I will keep your names and individual performance secret. I greatly appreciate your participation.

Please write your name: _____

Please pretend that you have returned to Joe Maxwell Internet Auctions website. Try to remember your password and write it in the box below.

Joe Maxwell Internet Auctions

Login

Password: [ ]

[Login]

Please log in again. If you are unsure of your password, please try a different one.

Joe Maxwell Internet Auctions

Login

Password: [ ]

[Login]

Now turn the sheet over and continue.

Please log in again. If you are still unsure of your password, please try a different one.

Joe Maxwell Internet Auctions

Login

Password: [ ]

[Login]

Please circle your answers to the following questions:

Did you remember your password?

yes probably don't know probably not no

Did you write your password on the questionnaires? yes no

Did you write your password somewhere else? yes no

How do you like your password?

hate it don't like it ok like it love it

How did you remember your password?

Was your password easy or hard to remember? Why do you think so?

Thank you for participating in this study of password generators. If you wish to receive a summary of the results, please write your email address: _____

Please return this sheet to Michael. Thank you.