Memorable 56-bit passwords using Markov Models and Huffman trees (and Charles Dickens)

DANNY BOGUS
Imaginary U.
1234 Example St.
Notastate, Noncountry
bogus@example.com

ABSTRACT

We describe a password generation scheme based on Markov models built from English text (specifically, Charles Dickens' *A Tale Of Two Cities*). We show a (linear-running-time) bijection between random bitstrings of any desired length and generated text, ensuring that all passwords are generated with equal probability. We observe that the generated passwords appear to strike a reasonable balance between memorability and security. Using the system, we get 56-bit passwords like `The cusay is wither?" t`, rather than passwords like `tQ$%Xc4Ef`.

Users can try the system, at

`https://www.example.org/BOGUS-URL/`

In order to verify that these passwords are more memorable than the obvious pick-characters-from-a-hat approach, we conducted a controlled experiment on participants in an upper-level college class over the course of five weeks. This experiment suggests that after several weeks of training, students were more likely to recall the passwords in the experimental group to within 20%.

1. INTRODUCTION

Users are very bad at choosing passwords.

In order to precisely quantify just how bad they are (and how much better we would like to be), we use the standard measure of "bits of entropy", due to Shannon (Shannon 1948). As an example, a password chosen randomly from a set of 1024 available passwords would exhibit 10 bits of entropy, and more generally, one chosen at random from a set of size S will exhibit $log_2 S$ bits of entropy.

In a system using user-chosen passwords, some passwords will be chosen more frequently than others. This makes it much harder to characterize the "average" entropy of the passwords, but analysis by Bonneau of more than 65 million Yahoo passwords suggests that an attacker that is content to crack 25% of passwords can do so by trying a pool whose size is 25% of $2^{17.6}$. That is, the least secure quarter of users are as safe as they would be with a randomly generated password with 17.6 bits of entropy (Bonneau 2012).

To see just how terrible this is, observe that we can easily construct a pool of 77 password-safe characters[1], so that a randomly generated password containing n characters will contain $n \log_2(77)$ or approximately $6.25n$ bits of entropy, and that the aforementioned 50% of users would be better served by a password of three randomly generated characters. To better gauge this difficulty, observe this set of 8 randomly generated e-character passwords:[2]

```
tBJ
fZX
evA
8Fy
MHr
=qe
f]w
YxU
```

We conjecture that most users could readily memorize one of these.[3]

Unfortunately, we need to set the target substantially higher. One standard attack model assumes that attackers will have access to encrypted passwords for offline testing, but that the password encryption scheme will use "key stretching," a method of relying on expensive-to-compute hashes in order to make checking passwords—and therefore, guessing passwords—more expensive.

Bonneau and Schechter suggest that under these constraints, and the further assumption that key-stretching can be increased to compensate for ever-faster machines, a password with 56 bits of entropy might well be considered adequate for some time to come (Bonneau and Schechter 2014).

The most straightforward way to achieve this goal is with randomly generated passwords. That is, users are assigned passwords by the system, rather than being allowed to choose their own. In fact, this was standard practice until approximately 1990 (Adams et al. 1997), when user convenience was seen to to trump security.

Today, the general assumption—evidenced by the lack of systems using randomly assigned passwords—is that users cannot be expected to recall secure passwords. Bonneau and Schechter (Bonneau and Schechter 2014) challenge this, and describe a study in which users were recruited for an experiment in which they were unwittingly learning to type a 56-bit password.[4] This experiment

[1] viz: abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890!^-=+[]@#$%&*()

[2] Throughout this paper, in the spirit of even-handedness and honesty, we have been careful to run each example only once, to avoid the tendency to "cherry-pick" examples that suit our points.

[3] Please don't use these passwords, or any other password printed in this paper. These passwords are officially toast.

[4] Later interviews suggested that some of them might have deduced

used *spaced repetition* (Cepeda et al. 2006; Ebbinghaus 1885), and found that users learned their passwords after a median of 36 logins, and that three days later, 88% recalled their passwords precisely, although 21% admitted having written them down.

2. HOW TO RANDOMLY GENERATE PASSWORDS?

If we're convinced that random passwords are a good idea, and that recalling a 56-bit password is at least within the realm of possibility, we must try to find a set of passwords (more specifically, a set of 2^{56} passwords) that are as memorable as possible.

We should acknowledge at the outset that there are many password schemes that use passwords that are not simply alphanumeric sequences, but include biometric data, 2-factor authentication, hardware keys, and the like. We acknowledge the work that's gone into these approaches, and we regard these schemes as outside the scope of this paper.

2.1 Random Characters

The first and most natural system is to generate passwords by choosing random sequences of characters from a given set, as described before. In order to see what a 56-bit password might look like in such a system, consider the following set of eight such passwords:

```
Ocd!SG3aU
)u)4OlXt%
tQ$%Xc4Ef
TH9H*kt7^
@f7naKFpx
K+UKdf^7c
S^UhiU#cm
usCGQZ)p-
```

In this system, a single randomly generated password has an entropy of 56.4 bits.

Naturally, a different alphabet can be used, and this will affect memorability. For instance, here we use an alphabet containing only one and zero:

```
11011111100111010101111100111010
 100010000110000011110110
10010110011110100010000011001111
 111000101100110010001001
```

In this system, each password is 56 characters long, and has exactly 56 bits of entropy. We conjecture that passwords such as these would be difficult to memorize. Also, we show only two such passwords, to save paper.

2.2 Random Words

Alternatively, many more than six bits can be encoded in each character, if we take as elements of our alphabet not single letters but rather words, or syllables.

The first of these, perhaps best known through the "Horse Battery Staple" XKCD comic (Monroe 2011), suggests that we use a word list, and choose from a small set of word separators to obtain a bit of extra entropy. Using the freely available RIDYHEW word list (Street 2003), we can obtain 18.8 bits of entropy for each word, plus 2 bits for each separator. In order to reach the 56-bit threshold, we must therefore use three of each, for a total of 62 bits of entropy. Here are eight examples:

```
reelman,phymas-quelea;
leapful;bubinga;morsures-
orientalised;liging-isographs-
molecule-charcoallier-foxings,
plaquette.cultivates.agraphobia-
mewsed;gasmasking;pech;
metencephalic.gulf.layoff;
kinematicses-pyknosomes;delineate.
```

Our observation (at the time of the comic's release) was that these sequences did not seem to be substantially nicer than the simple alphanumeric sequences, due in large part to the use of words like "pyknosomes," "quelea," and "phymas."

2.3 Random Syllables

A number of other schemes have attempted to split the difference between random characters and random words by using random syllables. One such scheme was adopted by the NIST (NIST 1993), although it was later found to be broken, in that it generated passwords with different probabilities (Ganesan and Davies 1994). Despite this, it is not difficult to devise a scheme in which all syllables are equally likely to be generated.

One example of such a scheme is given by Leonhard and Venkatakrishnan (Leonhard and Venkatakrishnan 2007). They generate words by choosing from a set of 13 templates, where each template indicates which characters must be consonants, and which characters must be vowels. So, for instance, one of the templates is "abbabbaa", indicating that the first character must be a vowel, the second two must be consonants, and so forth. Each consonant is chosen from a fixed set, as is each vowel. The resulting words have 30.8 bits of entropy; in order to achieve the needed 56, we can simply choose two of them.

Here are eight such examples:

```
kuyivavo rastgekoe
phoymasui nupiirji
ifstaezfa ihleophi
stifuyistu apibzaco
iholeyza gohwoopha
ebyexloi stustoijsto
maiwixdi enjujvia
dophaordu ostchichbou
```

3. DRIVING NONUNIFORM CHOICE USING BIT SOURCES

One characteristic of all of the approaches seen thus far is that they guarantee that every password is chosen with equal probability, using a simple approach. Specifically, password generation proceeds by making a fixed number of choices from a fixed number of a fixed set of elements.

Specifically, the first scheme generates a password by making exactly ten choices from sets of size 77, for all passwords. The last scheme is also careful to ensure the same number of vowels and consonants in each template, meaning that password generation always involves one choice from a set of size 13 followed by four choices from a set of size 5 (the vowels) and four choices from a set of size 22, followed by a second round of each of these (in order to generate a second word). For all of these schemes, every possible word is generated with equivalent probability. This property is crucial, since a system that generates some passwords with higher probability—such as the scheme adopted by the NIST (NIST 1993)—means that by focusing on more probable passwords, attackers can gain leverage.

the experiment's true goal.

This approach has a cost, though. In such a scheme, it is not possible to “favor” certain better-sounding or more-memorable passwords by biasing the system toward their selection; such a bias would increase the probability of certain passwords being generated, and thereby compromise the system.

3.1 Another Way

However, there is another way of guaranteeing that each password is generated with equal likelihood. If we can establish a (computable) bijection between the natural numbers in the range $[0, \ldots, N)$ and a set of passwords, then we can easily guarantee that each password is generated with equal probability by directly generating a random natural number, and then mapping it to the corresponding password.

In order to make such a scheme work, we must show that mapping is indeed a bijection, implying that no two numbers map to the same password.

3.2 Using Bits to Drive a Model

This idea opens up a new way to generate passwords. Rather than making a sequence of independent choices, we can build a model that draws randomness from a given sequence of bits. That is, we first generate a sequence of 56 random bits, and then use this as a stream of randomness to determine the behavior of a pseudo-random algorithm. If the stream of bits represents the only source of randomness, then in fact the algorithm is deterministic, and indeed determined entirely by the given sequence of bits.

Using this approach, we can lift the restriction (all local choices must be equally likely) that has dogged the creation of memorable or idiomatic-sounding password generators.

Specifically, our chosen non-uniform approach uses a Markov model, built from Charles Dickens’ *A Tale of Two Cities.* We conjecture that this choice is not a critical one.

4. MARKOV MODELS

In its simplest form, a Markov modelis simply a nondeterministic state machine. The model contains a set of states, and a set of transitions. Each transition has a probability associated with it, and we have the standard invariant that the sum of the probabilities of the transitions from the given states sum to one.

For our work, we built markov models from the sequences of characters[5] in Charles Dickens’ A Tale of Two Cities (Dickens 1859). One choice that we faced was how many characters to include in each state. For the sake of the following examples, we will fix this number at two.

To build the model, then, consider every pair of adjacent characters in the book. For instance, "ca" is one such pair of characters. Then, consider every character that follows this pair, and count how many times each occurs. This generates the distribution shown in figure 1:

In order to generate idiomatic text from this model, then, we should observe these distributions. That is, if the last two characters were "ca", the next character should be an "r" with probability 278/1397.

How should we make this choice? One way would be to draw enough bits (11) from our pool to get a number larger than 1397, and then, say, pick the letter "r" if the number is less than 278. Note, though, that while our program will be deterministic (since it gets its randomness from the stream of given bits), it will *not* represent a bijection, since (at least) 278 of the 2048 possible choices all go to the same state.

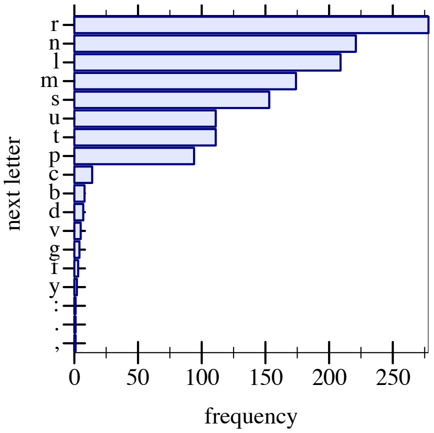


Figure 1: distribution of letters following"ca"

To solve this, we need a way of drawing fewer bits to make more common choices, and drawing more bits to make rarer ones.

Fortunately, this is exactly the problem that Huffman trees solve!

5. HUFFMAN TREES

Huffman trees (Huffman and others 1952) are generally used in compression. The basic idea is that we can build a binary tree where more-common choices are close to the root, and less-common choices are further from the root.

The standard construction algorithm for Huffman trees proceeds by coalescing; starting with a set of leaves with weights, we join together the two least-weighty leaves into a branch whose weight is the sum of its children. We then continue, until at last we’re left with just one tree.

As an example, we can consider the distribution given above. In this case, there are several characters (the comma, the period, and the colon) that occur just once. We would therefore combine two of these (the comma and the period, say) into a branch with weight two and two children, the comma and period leaves. Next, we would combine the colon (the only tree left with weight one) with either the "y" or the branch formed in the previous step; each has weight two. The result would have weight three.

Proceeding in this way, we arrive at the tree shown in figure 2.

If this tree were to be used in compression, we would represent the transition to the letter "r" using two bits, a zero and a zero (if we use zeros to denote left branches). The transition to the next most likely letter, "l", would be represented as one-zero-one. Note that less common choices are encoded using larger numbers of bits.

We are not interested in compression, but in generation. For this use case, we imagine that we are "decoding" the random bit stream. So, for instance, if the random bit stream contains the bits (0100110), we would use the first six bits to reach the leaf "c", and leave the remaining zero in the stream.

Once we’ve reached a character, we may add this character to the

[5] when we say characters, we mean letters in the alphabet, not the fictional subjects of the novel...

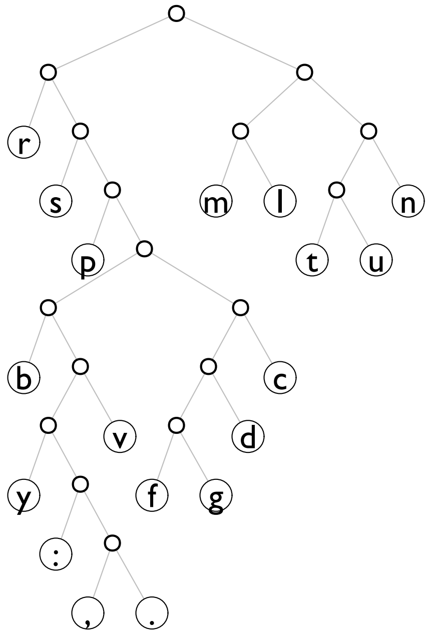


Figure 2: Huffman tree encoding next-letter choice from state "ca"

output stream. In order to continue, we must then start again, in the new state. If, for instance, the "l" were chosen, we would now be in the state corresponding to the letter pair "al", and we would begin again.

Consider once more the problem of proving that this is a bijection. In contrast to the earlier scheme, note that if two bit streams differ first at (say) bit n, then the character that is output at that point in the model's operation is guaranteed to be different. This ensures that each bit stream corresponds to a different output. To see the other half of the bijection, we observe that given a model's output, we can simply run the "compression" algorithm to obtain the sequence of bits that generated it.

5.1 Running Out of Bits

One minor complication arises in that the given scheme is not guaranteed to end "neatly". That is, the model may have only partially traversed a huffman tree when the end of the input bit stream is reached. We can easily solve this by observing the bijection between streams of 56 randomly generated bits and the infinite stream of bits whose first 56 bits are randomly generated and whose remaining bits are all “zero”, in much the same way that an integer is not changed by prepending an infinite stream of zeros. This allows us to implement a bit generator that simply defaults to “zero” when all bits are exhausted. In fact, the model could continue generating text, but there's no need to do so, since the 56 random bits have already been used.

5.2 The Forbidden state

Can our Markov model get stuck? This can occur if there is a state with no outgoing transition. Fortunately, the construction of the tree guarantees there will always be at least one transition... except for the final characters of the file. If this sequence occurs only once in the text file, it's conceivable that the model could get stuck. This problem can easily be solved, though, by considering the source document to be “circular,” and adding a transition from the final state to the file's initial character.

5.3 Choosing a Markov Model

In our examples thus far, we have chosen to use exactly two characters as the states in the Markov model. This is by no means the only choice. We can easily use one character, or three or four.

The tradeoff is fairly clear: using shorter character-strings results in strings that sound less like English, and using longer character-strings results in strings that more like English. There is, however, a price; the idiomaticity of the resulting strings results from a lower "compression", measured in bits per character. That is, the one-character markov model results in short strings, and the three- and four-character models result in longer ones. Naturally, all of the given models have the randomness properties we've shown for the two-character ones, and users may certainly choose a three- or four-character model, if they find that the increase in memorability compensates for the increase in length.

A final note concerns the selection of the initial state. We've chosen to choose from those states starting with a space, in order to simulate a password that begins "at the beginning of a word," and we build a huffman tree to choose from these initial states basen on their frequency within the text.

6. EXAMPLES

The proof is in the pudding! Let's see some examples.

First, we generate strings using the one-character Markov model:

```
sochete ftr d f
walowemfronlo
them-l parof h o
tacupis anemas a
ar ps o hen tsefr
adowepr,-ce he T
land tr slor. ter
my lly af a sioo
```

These may be seen to be short, but contain challenging sequences, such as `lly af a`.

Next, strings generated using the two-character Markov model:

```
witaing her or to soma
ronstionsay ragao
wiliking hus ands this st
in.'s.--overstichery
Driess, bursto anc
guavichfultakfull
way, Lounto coverb
Yah!--by be wings,--wi
```

These are slightly longer, but much more pronounceable, and appear somewhat more memorable.

Next, strings generated using the three-character Markov model:

```
younde; a mad revide s
thround eignal coff his, m
he who's rests. The off
freets, Mr. Befolks our chr
not on the said Midn't hest
of peak out it, an off (m
were walls. Twice that. It i
```

```
know thin one be thing; i
```

These are far more English-like, with many actual words. As a side note, the phrases generated here and in by the prior two-character model appear almost archaic, with words like "younde," "coff," and "hest". Naturally, these are longer than the prior set.

Finally, strings generated using the four-character Markov model:

```
naughed, Who tall her o
in them forturbatious, who I knew p
Fancy? News of all? Two. If than
growing lumbent if thankle imp
commonsteady for heavy had hom
a luncher's sacrificers was. T
kiss Manettle clothed them, beni
naminished--this. What? Oh! It wi
```

At this point, it's starting to become clear what the source is, and in some strings, Sydney Carton appears by name. In addition, you get some fairly interesting neologisms—in this case, "forturbatious." It's not a word, but maybe it should be.

7. CHOICE OF CORPUS

Naturally, the choice of *A Tale of Two Cities* is largely arbitrary; any corpus of reasonable length will suffice. One intriguing possibility would be to choose the full text of all of the e-mails in a particular user's history.[6] This text would presumably reflect the style of text that a particular user is accustomed to read and write, and should in principle be extraordinarily memorable. Note that the security of the system is entirely independent of the chosen corpus; our attack model assumes that the attacker already has the full text of the corpus.

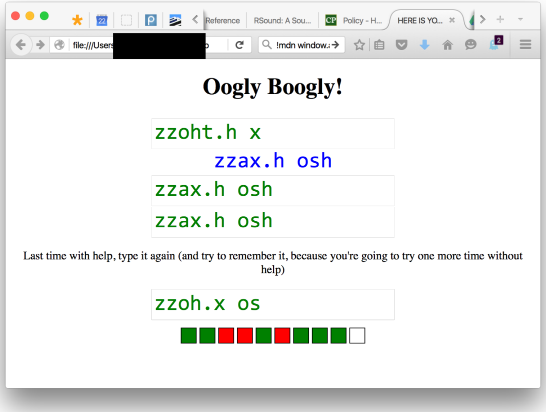
8. EVALUATION

In order to explore the advantages of our proposed system, we designed and executed an experiment.

8.1 Subjects and Procedure

We recruited 70 students from an upper-level university class to participate in a one-minute training session at the beginning of an in-class lab, each time it met. These sessions occurred three times a week, and the experiment ran for a total of 13 sessions. The students were randomly assigned to either the experimental group or to the control group. Each student was assigned a single password to be learned during the course of the experiment. Those in the experimental group were assigned a password generated using our system, set to use an order-two model generated from *A Tale of Two Cities*. Those in the control group were assigned a random nine- or ten-character password generated by choosing randomly from a set of characters. Both the experimental and the control passwords were generated using 56 bits of randomness each.

At the beginning of each lab session, students visited a web page that required them to log in using existing student credentials, and then asked them to type the password that they'd previously been given. (Naturally, the first time they used the system, their guesses were entirely incorrect.) Following this, they were shown the correct password and asked to type it three times with the assistance of green and red squares indicating whether the corresponding character had been typed correctly.



Oogly Boogly!

zzoht.h x

zzax.h osh

zzax.h osh

zzax.h osh

Last time with help, type it again (and try to remember it, because you're going to try one more time without help)

zzoh.x os

🟩🟩🟥🟥🟩🟥🟩🟩🟩□

After three assisted attempts, students were again challenged to type the password without being able to see it. They were then finished, until the next training session.

During the student's interaction, the system logged each user's session start, and every change to a password entry box, along with timing information. In essence, the system acted as a keylogger.

Note that no active deception was involved in the experiment; students knew that they were participating in a study about the memorability of passwords. The experiment's plan was reviewed by our Institutional Review Board, and deemed "exempt from further review."

8.2 Pre-Registration

While the experiment was going on, we discovered the existence of the Open Science Foundation, online at `https://osf.io/`. Their mission is to help ensure the quality of experimentation in the natural sciences by allowing experimenters to describe the experiments that they are performing and the analyses that they plan to perform *before* examining the data and extracting the hypotheses that are best supported by the data ("hmm, it looks like passwords containing exactly three spaces are much more memorable!"). We registered our project, providing pointers to code, and a plan for analysis (BOGUS 2016).

In our pre-registration, we described two hypotheses. The first was that our passwords would be learned more quickly, and the second was that they would be retained longer.

Additionally, we decided to omit the information of any student that participated fewer than three times.

8.3 Analysis

In order to measure password learning, our primary instrument was the password entered by each student into the first, unprompted, password box that was a part of each session. We used Levenshtein string distance (Levenshtein 1966) as a measure of password correctness. This metric measures the number of one-character changes—insertions, deletions, or substitutions—that are necessary to change one string into another. So, for instance, if the student omitted one character and replaced an 'a' with an 'e' but was otherwise correct, the Levenshtein distance would be computed as two. We then divided this by the number of characters in the password to obtain a measure of error that ranges from 0.0, representing a correct password entry, to 1.0, representing an entirely wrong password.

[6] This is actually not hypothetical; we do exactly this in the generation of our own passwords.

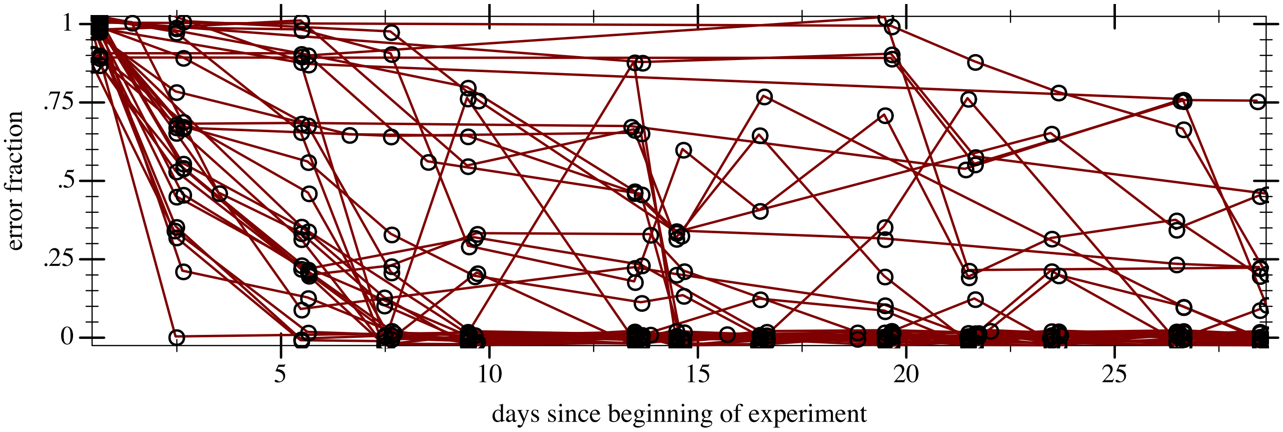


Figure 3: performance of control group in first box

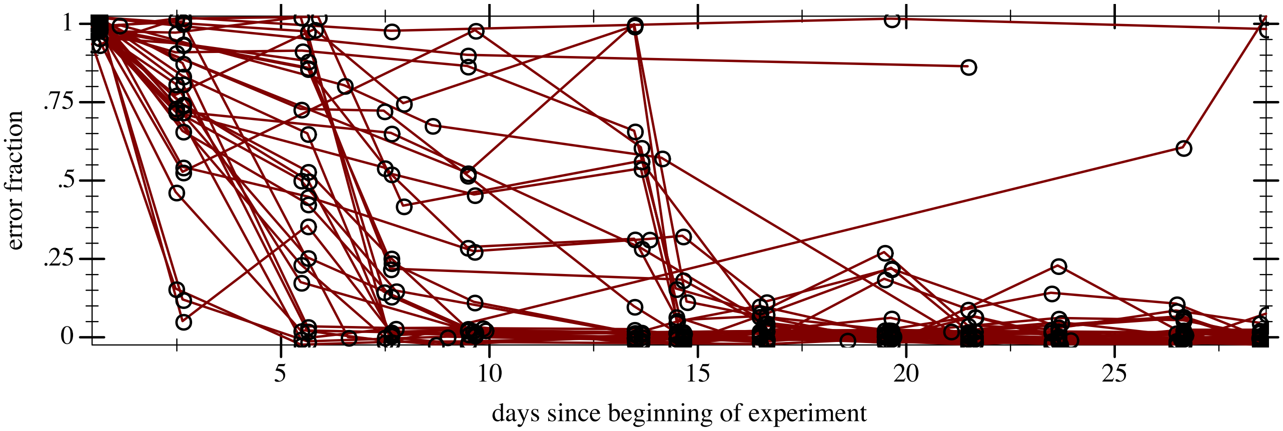


Figure 4: performance of experimental group in first box

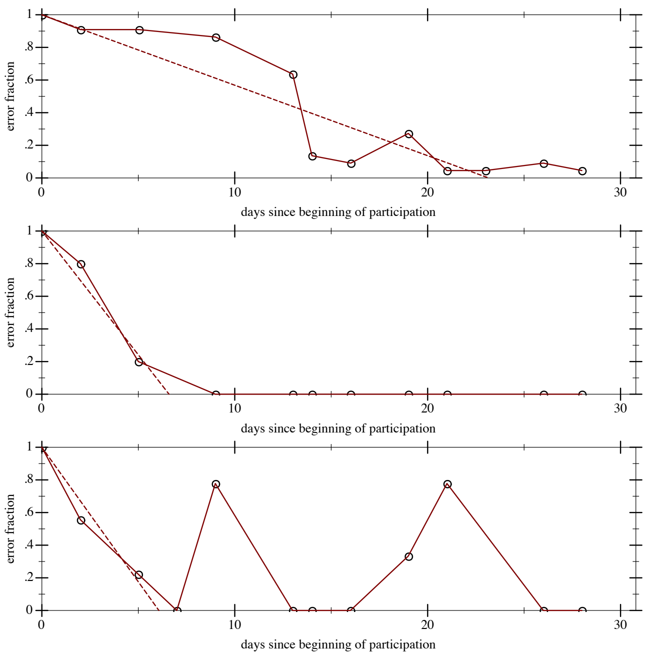


Figure 5: student traces and best-fit model for three students

Figure 3 shows the behavior of the students in the control group in the first box from each day's training. Figure 4 shows the behavior of the students in the experimental group in the first box from each day's training. Each point corresponds to a single student's session. In these graphs, each error value was perturbed by a random value in the range $(-0.025, 0.025)$, in order to prevent the preponderance of points precisely on the x axis from all obscuring each other.

A quick glance at these two figures suggests that there is in fact a difference between the two groups. But is it the difference that we hypothesized?

8.3.1 Not Really, No

Our first hypothesis was that the experimental passwords would be learned more quickly than the control passwords.

In order to measure this, we fitted a simple two-part linear model to each student's performance. The model is parameterized by a single parameter, a "learning time," l. The model is then defined by the functions f_l:

$$f_l(t) = \begin{cases} 1 - t/l & \text{when } t < l \\ 0 & \text{otherwise} \end{cases}$$

For each student, we chose the value of l that minimized the mean-square distance between the sample points and the fitted line segments.

Figure 5 shows examples of the best-fit model for three different students. The first one learns slowly, with an estimated time of about 23 days to learn. The second (more typical) one learns quickly, with an estimated time of about 6.5 days to learn. The third one illustrates one of the problems with the model; the best fit shows that learning took 6 days, despite clear problems in recall in later experiments.[7]

Noisy data notwithstanding, we computed a value of l for each member of the control and experimental group. Figure 6 graphs the density of the values of l for each group. That is, we convolve a graph of impulse functions with normal kernels to show a "smoothed" density graph. Each group's density curve has an integral of 1.[8] The control group is shown with a dashed line, and the experimental group with a solid one.

Omitted from this graph is one subject who entered his or her password incorrectly with perfect consistency, leading to an estimated learning time of infinity.[9]

Next, note the scale of the x axis; The experiment lasted for 28 days, and the rightmost entry on this density graph lies at about 168 days.

Our first observation is that the two sets are not normally distributed by any stretch of the imagination. Attempting to perform a student's t-test, for instance, would be relatively meaningless. Each one has a remarkably strong density in the five-to-six day range, with a number of outliers. Specifically, approximately one sixth of the participants have values of l greater than 28 days, the length of the experiment, indicating essentially that they do not appear to have learned their passwords.

Next, inspection suggests that many of the outliers are students that participated in a relatively small number of trials. Naturally, we would expect these students to take longer to learn their passwords.

Returning to our first hypothesis: can we conclude that students learn the experimental passwords more quickly? No, we cannot. Both distributions appear to have a heavy tail, and in fact, the control group has a stronger representation among the very quickest learners.

8.3.2 Retention

Our second stated hypothesis was that students would retain their passwords better in the experimental group. The final element of this experiment—not yet completed—is a delay of two weeks, after which the students will be asked (in the context of the class final exam) to produce their password (on an anonymous slip of paper). This will provide a measure of password retention. As a side benefit, it may also help to identify those that were cheating by recording their password in digital form.

8.4 Data Mining

Building post-hoc hypotheses is a big no-no. This is called "p-hacking," based on the observation that for a given set of data, it is nearly always possible to tweak and grind the hypothesis in order to obtain statistical significance.

What follows, then, is not a test of a hypothesis, but rather simply a set of observations that might lead to further hypotheses.

Our inspection of the graphs in figure 3 and figure 4, along with our analysis of estimated learning times, suggests that the control group contains a substantial subgroup that has not learned their passwords at all. In order to measure this, we graphed, for various error levels,

[7] The fit of this model is unintuitive; it appears that moving the intercept to the right would produce a better fit, given the penalty associated with the later points. Curiously, this is not the case.

[8] We adopt this alternative to a histogram in order to avoid the inevitable bias that occurs as a result of the selection of bin boundaries.

[9] Or, more precisely, to divergence of our estimation algorithm.

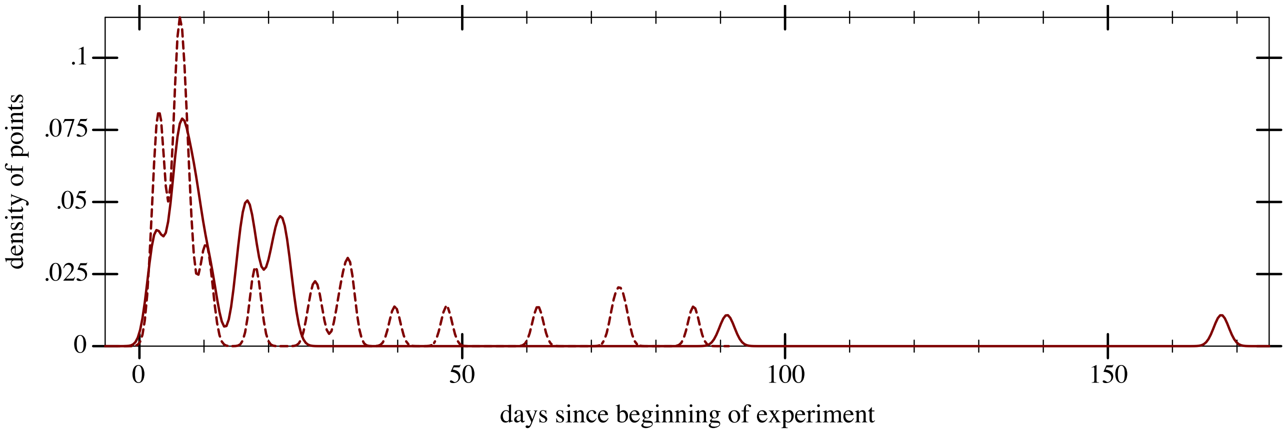


Figure 6: estimated learning times (control group dashed lines)

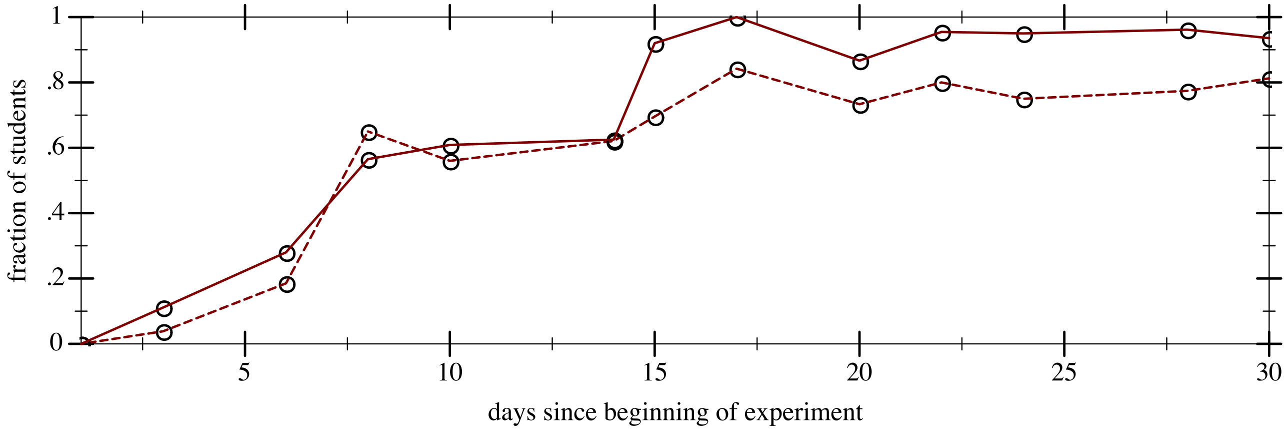


Figure 7: fraction of students with less than 20% error (control group dashed)

the fraction of students in each session whose error was below the specified level. Figure 7 shows the fraction of students that had an error level of less than 20% in a given test. As before, the dashed line indicates the control group.

In this graph, it appears that a significantly larger fraction of the students in the experimental group recalled their passwords, but only after an initial learning period of approximately 6 sessions. The graphs for 10%, 30%, and 40% are similar. It would be relatively straightforward to attempt to validate this with a second experiment that focuses on this aspect of the project.

8.5 Threats to Validity

Naturally, any study with live subjects is subject to threats to validity both internal and external.

One potential internal threat to validity arises as a result of possible cheating. It its certainly possible for students to copy and paste the passwords from the web page. We do not see much evidence of this, both in that only one student appears to have learned the password after one session, and also in that our keystroke data shows only one instance in which the password "appeared" in exactly one keystroke. Additionally, the experimental setting (in lab) should help to dissuade students from cheating. Finally, we see no reason to expect the experimental group to cheat more than the control group, or vice versa.

Another potential internal threat to validity concerns the frequency of participation. Some students participated in every session, and others only in a few. A tighter distribution might arise from an experiment that required all students to participate in all sessions.

An external threat to validity arises from the makeup of the experimental population. As upper-level computer science students, it's reasonable to suspect that these subjects will be much better at learning passwords than the population at large. However, we do not think that this will affect the relative performance of the control and experimental groups.

9. REPRODUCIBILITY

We have made every effort to ensure that our work is entirely reproducible, making available all code involved in password generation and experimentation, and also the raw (anonymized) data collected during the experiment.

Firstly, the code for the password generation algorithm is freely available on github, at `https://www.github.com/EXAMPLE`.

Next, the code for the web application used to gather student data is also available as part of a different github repository, at `https://www.github.com/EXAMPLE`.

Finally, the raw keystroke-level data (post-anonymization) is available as a compressed log file, at `https://www.example.com/BOGUS-URL`.

10. RELATED WORK

There are many, many works that describe passwords. We have cited Bonneau's work before, and we will do so again here, as this work was enormously informative (Bonneau and Schechter 2014). We have also already described the work contained in many other related projects (Leonhard and Venkatakrishnan 2007; NIST 1993).

To our knowledge, however, there is no other work that uses a bit source to drive huffman decoding to drive a markov model, thereby enabling generation of pronounceable text without the (heretofore) attendant lack of equi-probability.

11. FUTURE WORK

One of the biggest issues in constructing this experiment concerned the choice of a set of passwords for the control group. We elected to use what we saw as the simplest baseline—random characters—but we acknowledge that it might also be useful to compare our proposed system to one such as "horse battery staple." (Monroe 2011)

Also, now that we've conducted one experiment, we are in a position to formulate more precise hypotheses. Specifically, it appears that the proposed system produces a higher likelihood of eventual success, rather than promoting faster learning. Perhaps the clearest method of testing this would be to perform a paired test, where each student was required to learn both a string from the control set and one from the experimental set.

Finally, another potential use of the system, leveraging the human ability to rapidly ingest written text, is in the generation of fingerprints for encryption keys.

As an example, here's the SHA1 fingerprint of a certificate, represented as a hexadecimal string:

```
46:DC:1D:39:45:88:2A:6B:90:D2:AC:9E:0A:81:
5E:9A:33:26:03:B1
```

... and here's the same fingerprint, represented as a molis hai string (using the same order-2 model associated with the experiment):

```
on, satinusibe his saingespards blin fevis not
not beavi
```

We conjecture that a user is much more likely to be able to rapidly compare fingerprints expressed in this way.

Further experimentation required!

12. ACKNOWLEDGMENTS

Many thanks to REDACTED for instantly pinpointing relevant research. Many thanks to REDACTED for being an amazing programming language. Many thanks to REDACTED, REDACTED, and REDACTED for helpful discussions on how best to analyze the data.

Bibliography

Anne Adams, Martina Angela Sasse, and Peter Lunt. Making passwords secure and usable. In *Proc. People and Computers XII*, pp. 1–19, 1997.

DANNY BOGUS. Molis Hai Memorability. at URL: https://nsf.io/EXAMPLE, 2016.

Joseph Bonneau. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *Proc. 2012 IEEE Symposium on Security and Privacy*, 2012.

Joseph Bonneau and Stuart Schechter. Towards reliable storage of 56-bit secrets in human memory. In *Proc. Proc. USENIX Security*, 2014.

Nicholas J. Cepeda, Harold Pashler, Edward Vul, John T. Wixted, and Doug Rohrer. Distributed practice in verbal recall tasks: A review and quantitative synthesis. *Psychological Bulletin* 132(3), 2006.

Charles Dickens. A Tale of Two Cities. Chapman & Hall, 1859.

Hermann Ebbinghaus. Über das gedächtnis: untersuchungen zur experimentellen psychologie. Duncker & Humblot, 1885.

Ravi Ganesan and Chris Davies. A new attack on random pronounceable password generators. In *Proc. Proceedings of the 17th NIST-NCSC National Computer Security Conference*, 1994.

David A. Huffman and others. A method for the construction of minimum redundancy codes. *Proceedings of the IRE* 40(9), pp. 1098–1101, 1952.

Michael D. Leonhard and VN Venkatakrishnan. A comparative study of three random password generators. *IEEE EIT*, pp. 227–232, 2007.

Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady* 10(8), 1966.

Randall Monroe. Password Strength, XKCD #936. at URL: https://www.xkcd.com/936/, 2011.

NIST. Automated Password Generator. Federal Information Processing Standards Publication No. 181, 1993.

Claude E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal* 7, pp. 379–423, 1948.

Chris Street. RIDYHEW. The RIDiculouslY Huge English Wordlist. at URL: http://www.codehappy.net/wordlist.htm, 2003.