

# Generating 56-bit passwords using Markov Models (and Charles Dickens)

John Clements

Cal Poly San Luis Obispo  
clements@brinckerhoff.org

## 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`.

## 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<sup>1</sup>, 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 diffi-

culty, observe this set of 8 randomly generated e-character passwords:<sup>2</sup>

tBJ  
fZX  
evA  
8Fy  
Mhr  
=qe  
f]w  
YxU

We conjecture that most users could readily memorize one of these.<sup>3</sup>

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 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.<sup>4</sup> This experiment 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.

<sup>1</sup> viz: abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTU-VWXYZ 1234567890!^-=+[]@#%&\*()

<sup>2</sup> 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.

<sup>3</sup> Please don't use these passwords, or any other password printed in this paper. These passwords are officially toast.

<sup>4</sup> Later interviews suggested that some of them might have deduced the experiment's true goal.

## 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. 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)40lXt%
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, we use an alphabet containing only one and zero:

```
11011111100111010101111100111010100010000111000011110110
10010110011110100010000011001111110001011001100110011001
11101101110001000001011001011110000111000101100000011101
111010010000110101001100100110001110001100111100000011101
001101100110011100110001110011111101101101010011111000
11001001101011110111101010100100010001110111111111111111
1000010111011110010101011101000011111011101011011111111
111010101000100110100001010100001010101100101101110001001
```

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.

### 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 ???), 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-charcoalier-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 nupiiirji
ifstaezfa ihleophi
stifuyistu apibzaco
iholeyza gohwoopha
ebyexloi stustoiijsto
maiwxdi 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.

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  $[0, \dots, N)$  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.

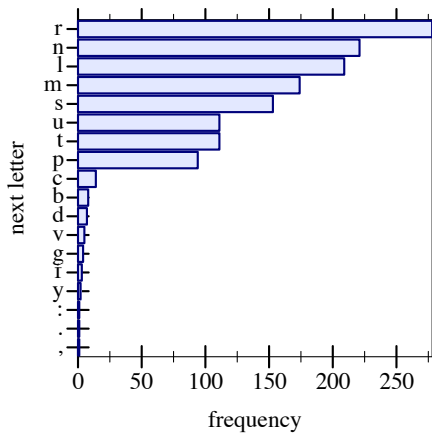


Figure 1: distribution of letters following "ca"

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 (apparent) nondeterminism, 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 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 model is 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<sup>5</sup> 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

<sup>5</sup> when we say characters, we mean letters in the alphabet, not the fictional subjects of the novel...

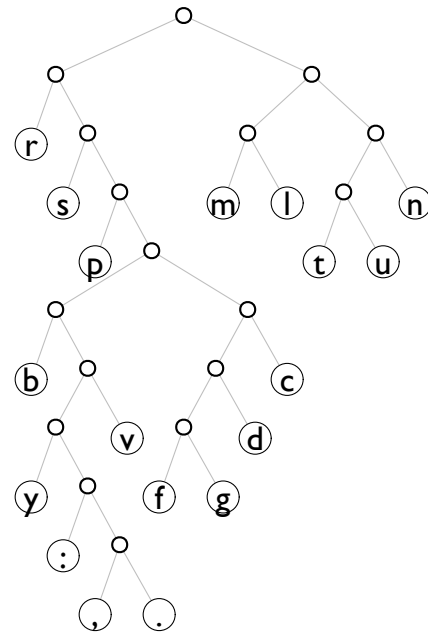


Figure 2: Huffman tree encoding next-letter choice from state "ca"

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.

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 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 simply to start with the appropriate-length substring of "The

". Naturally, the starting state could be chosen at random, to obtain slightly shorter strings.

## 6. Examples

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

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

```
Tenon thempea co ts
Te od " perdy, wil
Thalivares youety
T.) reait dean,
Tr, 'ser h Lof owey
Tempr." gedolam,
Te cty se d y Mr,-
Tere th, Fand ry."
```

These may be seen to be short, but contain challenging sequences, such as `cty se d y`.

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

```
Therfur, unappen. So
Therying hant abree,
The cusay is wither?" t
The greed hispefters and
The as obe so yon ters
Thad gre strow; agamo
Thereakentin town ing." "MO
Their, anytel' hat," "te"
```

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

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

```
Ther highly to a vice of eart
Then," suspeakings beers ways
They, anythis, int found mad
They?" "If, who waite any," mul
The moritiour him; businenl
Thensuspellectiver fur
Then him do nown wilty," res
The fix, buse hand, followest."
```

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 "waite," "nown," and "yon". Naturally, these are longer than the prior set.

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

```
The tile fareweloped, and ever p
The shing it nother to delve w
The found, Sydney Carton wreckles Evremonds.
Su
The snorting ever in by turbed t
The receive a year." To appeality a
The back understitch ther's the
The wrong and, here!"--Mr. Calm info
The diffidelicitizen aparticulous timo
```

At this point, it's fairly clear what the source is; Sydney Carton appears by name. In addition, you get some fairly interesting neologisms—in this case, "diffidelicitizen." It's not a word, but maybe it should be. Also, we see some people that aren't actually in the book, including "Mr. Calm info."

## 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. 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. 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.

## 9. Future Work

There's a giant piece of future work here: specifically, we wave our hands and suggest that our passwords are more memorable than those generated by other schemes. Naturally, a claim like this cannot simply be taken as true; we must conduct a test to verify this claim.

We are currently building the tools to allow us to conduct this study.

## Bibliography

- Anne Adams, Martina Angela Sasse, and Peter Lunt. Making passwords secure and usable. In *Proc. People and Computers XII*, pp. 1–19, 1997.
- 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.
- Randall Monroe. Password Strength, XKCD #936. on the web: <http://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. on the web: <http://www.codehappy.net/wordlist.htm>.

<http://arxiv.org/abs/1502.07786>