Kwyjibo: automatic domain name generation

Heather Crawford and John Aycock* †

Department of Computer Science, University of Calgary, 2500 University Drive NW, Calgary, Alta., Canada T2N 1N4

SUMMARY

Automatically generating ‘good’ domain names that are random yet pronounceable is a problem harder than it first appears. The problem is related to random word generation, and we survey and categorize existing techniques before presenting our own syllable-based algorithm that produces higher-quality results. Our results are also applicable elsewhere, in areas such as password generation, username generation, and even computer-generated poetry. Copyright © 2008 John Wiley & Sons, Ltd.

Received 27 August 2007; Revised 13 December 2007; Accepted 18 February 2008

KEY WORDS: domain names; random word generation; hyphenation; Internet

1. INTRODUCTION

We have recently built a specialized Internet simulator for security teaching and research. Initially used in our unique course on spam and spyware [1], the simulator provided Web servers and automatically generated Web pages, a Web search engine and Web directory, proxy servers, and mail servers. Students browsed the Web pages using a normal Web browser, or ran arbitrary code to interact with the servers. The simulator thus had to meet not only a variety of technical constraints, but also had to induce a certain degree of suspension of disbelief for students using the simulated Internet directly.

One of the first issues we had to address in the simulator was how to generate domain names. How hard could it be? Generating domain names was more challenging than we anticipated, as it turned out, and it has taken a great deal of experimentation to get right. We had six criteria for a

*Correspondence to: John Aycock, Department of Computer Science, University of Calgary, 2500 University Drive NW, Calgary, Alta., Canada T2N 1N4.
†E-mail: aycock@ucalgary.ca

Contract/grant sponsor: Natural Sciences and Engineering Research Council of Canada

Copyright © 2008 John Wiley & Sons, Ltd.
good solution:

1. The generation algorithm must be able to produce at least one million unique domain names. This is the number of domains we targeted in our simulator. This does not necessarily mean that every domain name produced must be unique, just that the algorithm is capable of generating this many. The algorithm may be called repeatedly until it yields a unique name.
2. The domain names cannot be ‘gibberish.’ For example, we could take the MD5 hash of all the numbers between 1 and 1,000,000, resulting in domain names such as

   a1d0c6e83f027327d8461063f4ac58a6.com

   but humans need to work with the domain names; hence, domain names conducive to this are preferred. By itself, this could simply imply that the domain names be short.
3. The domain names cannot be easily guessed. Domain names of the form x00001.com, x00002.com, x00003.com are trivially generated, for instance, but they are also trivially guessed. The real Internet is discovered through crawling, and permitting our simulated Internet to be enumerated rather than crawled would represent a serious design flaw.
4. Ideally, the domain names should be pronounceable by native English speakers, again towards making them easy for (English-speaking) humans to use.
5. The generator must be run without Internet access, because it runs in a secure lab with an isolated network.
6. No trademarked names should intentionally appear in the domain names. This precludes harvesting existing domain names from Web sites and other Internet sources. Although it is not generally possible to anticipate and avoid all trademarked names, given the constraints on our solution, we can at least make a good-faith attempt to avoid sources of potentially trademarked names.

Besides our Internet simulation, the domain name generator would also be potentially of interest to real domain name registrars, some of whom suggest unregistered domain names.

There are other possible applications. The name generator could be used to produce pronounceable, non-dictionary words as default passwords for user accounts, or as challenges for CAPTCHA systems [2]. It could also be used to supply a stream of trademarkable names for new products. The second author used it to choose a username for an online service, after spending much time trying to manually pick a username that had not been taken. More far-ranging are applications to computer-generated poetry (e.g. [3]) to automatically produce poems such as Lewis Carroll’s Jabberwocky.

In the following sections, we describe our algorithm for generating domain names.

2. DOMAIN NAME GENERATION

To be precise, we are concerned with second-level domain names, the ‘xxx’ in xxx.com. Top-level domain names, such as .com and .org, can be easily chosen and appended onto a generated second-level domain name. We will use ‘domain name generation’ in this paper to mean second-level domain name generation, for simplicity.
What we want for domain name generation is an algorithm that produces a relatively short, pronounceable word. Choosing dictionary words at random is an obvious approach, but a large number of these coincide with real, possibly trademarked, names. (A good party game is to try and guess a dictionary word that is not registered as a .com domain.) Instead, we would like a randomly generated word, not a randomly selected word.

Random word generators exist, but it does not appear that the techniques have been collected and described anywhere in the literature. Some are ill-described, and implemented in proprietary code. The best-documented instances are used for producing random, yet pronounceable, passwords; others are billed as name generators, such as names for characters in role-playing games. We divide these generators into four types.

Selection-based: Selection-based generators are equipped with tables of strings, where each string is a portion of a word, possibly but not necessarily a syllable. The generator makes random selections from these tables to construct a word. An online generator using this technique has 10 tables, ranging in size from 20 to 153 strings. Strings from two, sometimes three, of these tables are pieced together, yielding words such as helioliga arcaden panend giganation polyrure

(The examples from generators used throughout this paper are not hand selected, but are simply the first $N$ words in the generated list, or the results of $N$ consecutive runs of the generator if a list is not produced.)

A generalization of selection-based generators is where a user manually specifies a template; this template controls how words are generated. Details of the template are obviously dependent on the implementation, but one example allows the placement of vowels, consonants, and combinations thereof as well as syllables. All the vowels, consonants, and syllables are chosen from manually constructed tables. Selection-based generators are naturally limited by the number of possible combinations and the imagination of the human programming them.

Transposition-based: A transposition-based generator takes dictionary words and transforms them according to some set of rules, where the rules are manually chosen to preserve pronounceability. For example, one product substitutes vowels, such as replacing ‘a’ with ‘e,’ then cleans up by replacing double vowels not normally seen in English with single vowels. Some of their generated words are grespir huend brevist seblit shrolly

More elaborate rules are possible that take a letter’s context in account, such as replacing ‘str’ with ‘spr.’ However, our experiments to automatically find such substitutions did not produce good results. We conjecture that, because pronounceability is highly sensitive to context, it may not be possible to extend this technique without encoding pronunciation rules into the generator.

Syllable-generating: Some random word generators encode rules of pronunciation, then use those rules to randomly construct pronounceable syllables; those syllables can then be concatenated to form a random word. Extra rules may be applied to avoid problems where pronounceable syllables become unpronounceable when placed consecutively. These ideas date back to the early 1970s.

‡In this paper, we use ‘dictionary’ in the usual computer security sense to refer to a list of words.
An implementation of the FIPS 181 syllable-generating algorithm [12] produces disappointing results that are not easily pronounced by native English speakers:

avubfo omdres ebtoktr irechpep ajcroadi

Another implementation [13] fares no better:

hikagroj nacussi iathlu dykidowx ampunk

$m,n$-gram: In this type of generator, the previously generated $m$ characters are used to select the $n$ characters to be generated next. Although, strictly speaking, it is not necessary to do so, the implementations of these generators invariably rely on statistics: a dictionary is used to automatically determine the frequency with which each distinct $n$-gram follows a particular $m$-gram. A Markov process can then be used to generate random words. Similar to syllable-generating methods, this technique has a long history: Multics reportedly had a random password generator based on ‘English digraph statistics’ [14, p. 394].

The results? A 2, 1-gram generator [15] produces words that are good but not great, on the basis of their difficulty to pronounce:

asspalat xicscosa ismelism roncungl torousha

A 2, 2-gram generator [16] fared the best of the existing techniques, although it was somewhat inconsistent in generated word quality:

masts radepeous neariarsarmigration shiffestraisleencistemy
gatiness litv tived aplaced ard sharanny

The value of $n$ is an important design choice. Our own experiments and other accounts [17,18] confirm that too large an $n$ causes the generator to regurgitate a lot of the original training input verbatim. This is desirable if the goal is to produce English text or dictionary words, but not for our goal of producing non-dictionary words.

Our approach to random word generation produces consistently good results, requires very little code to implement, and needs no manual tuning or configuration. Looking at other techniques, the idea behind syllable generation—concatenating pronounceable syllables together—is intuitively appealing, but the results of automatically generating syllables are clearly lacking. On the other hand, the automatic training of the $m,n$-gram approach is desirable, but the use of $m$ and $n$ characters is artificial and does not correspond to any property of pronounceable words.

We combine the two approaches in our algorithm and introduce a fifth type: $m,n$-syllable. Our implementation automatically learns the frequency of one syllable following another by breaking dictionary words into syllables using a time-tested hyphenation algorithm [19]$^\S$. A Markov process is then used to randomly generate words. Figures 1 and 2 give pseudocode for 1, 1-syllable training and word generation, respectively. The results obtained by choosing $m=n=1$ were sufficiently good that we did not need to exceed these values in practice.

During training, we filter out words of less than two syllables, and words that have excessively long or short syllables. (We use $\text{MINSYLLLEN}=2$ and $\text{MAXSYLLLEN}=4$.) Words’ start syllables and

$^\S$This is the hyphenation algorithm used by \LaTeX{} [20, Appendix H].
def train():
    for each dictionary word W:
        sylls = hyphenate(W)
        if len(sylls) < 2:
            continue
        if len(s) < MIN_SYLL_LEN for some s ∈ sylls:
            continue
        if len(s) > MAX_SYLL_LEN for some s ∈ sylls:
            continue

    START_SYLL = START_SYLL + sylls[0]
    END_SYLL = END_SYLL + sylls[len(sylls)-1]

    for i in [0...len(sylls)-1]:
        syll = sylls[i]
        if i > 0:
            last = sylls[i-1]
            S[last] = S[last] + syll

    for each syll in END_SYLL:
        if syll occurs less than 0.1% of the time:
            delete syll from END_SYLL

def hyphenate(W):
    #
    # Hyphenates W using Liang's algorithm,
    # and returns a list of syllables.
    #

Figure 1. Pseudocode for training the 1,1-syllable generator.

end syllables are appended onto the corresponding lists START_SYLL and END_SYLL; duplicates are not removed when appending onto lists; hence, frequency data are preserved. S is a hash table, keyed by syllables, where the value associated with a key is a list of syllables that follow in some dictionary word. For brevity, we assume that indexing S using a non-existent key returns an empty list.

We found that a judicious choice of end syllable was important for generating good-quality pronounceable words, and that is done three ways in the training and generation code. First, we filter out rarely occurring end syllables from the dictionary words, as they may be less familiar and harder to pronounce. Second, we ensure that a generated word ends with an end syllable found in some dictionary word. Third, choosing a syllable during generation that was only seen at the end of a dictionary word causes word generation to be exited prematurely.

Word generation itself is straightforward, starting with a syllable seen to start some dictionary word, and choosing follow-on syllables from S. Subject to the end syllable constraints above, we generate between 2 and MAX_SYLL=4 syllables in a word; the exact number is randomly chosen. Although the infinite loop in the generator may appear alarming, the code has never failed to terminate even after millions of runs, and in fact must terminate unless there is a fatal flaw in the random number generator.
The resulting words, excluding dictionary words, look like

beanfesting suinglying rhodangle underin anthrocyclic
sual underbidness interfensive arterier unbuckishly
sulcateness lulosematic sulate digressioner anaculturis

The dictionary in this case comprised 355 545 lowercase words from /usr/share/dict/words on Scientific Linux SL release 4.1. Some easy modifications to the generation algorithm include ensuring that a generated word does not appear in the dictionary, filtering out words with potentially offensive substrings, and guarding against repeated syllables in a word (as in suinglying).

3. CONCLUSION

Large numbers of names can be automatically generated that are randomly selected, yet seem to retain a high degree of pronounceability and usability for humans. As future work, we would like to extend our result to languages other than English, and derive an objective measure of the ’goodness’ of a generated name.
The name-generation method we describe here produces better-quality results than other techniques, techniques that have not all been gathered together and described in the literature until now. These automatically generated names are used as the basis for producing domain names, in our case, although they could also be used for other applications, such as generating pronounceable passwords.

ACKNOWLEDGEMENTS

The second author’s work is supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada. Thanks to Morrie Gasser, Richard Lowe, Richard Lawrence, and Sam Stoddard for answering questions about their random word generators. The anonymous referees made many comments that improved the presentation and focus of this work.

REFERENCES

1. Aycock J. Teaching spam and spyware at the University of C@lg4ry. Third Conference on Email and Anti-Spam (CEAS 2006), short paper, Mountain View, CA, 2006; 137–141.
4. NOEMATA Name Generator: http://noemata.net/nbng, c. 2002. [26 June 2007].