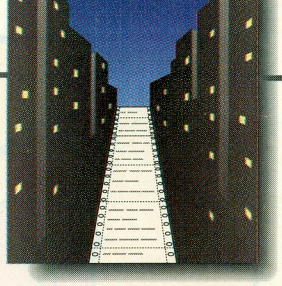


Password Generation by Bloom Filters

William Stallings

Introduction
by Bruce Schneier

Niklaus Wirth said: "Algorithms+data structures=programs." Every computer program consists of complex algorithms: to sort the database, compute the formulas, draw the graphics, and display the data. Often, the difference between a good program and a bad one is the underlying algorithms.

"Algorithm Alley" explores the design and implementation of algorithms. Every month I'll present useful algorithms that you can implement today. The algorithms will cover a variety of areas — computation, graphics, databases, networking, artificial intelligence, and more — and be relevant to many more applications.

The ultimate goal of this column is to help you think about algorithms so you can develop your own. A craft so varied as programming cannot be taught as a series of recipes. No matter how many algorithms I present, you're going to need something else. If I can teach you general principles of algorithms, then you can take them with you wherever you program.

My first column is about Bloom filters, a method of hashing that greatly reduces memory requirements at the expense of false "hits." They are useful in a variety of applications, particularly those in which no calculation is required if the search is unsuccessful. For example, you might want to check someone's credit rating or passport number, but do nothing else if the record doesn't exist. While Bloom filters will occasionally report that a record exists when it doesn't, they'll never erroneously report that a record doesn't exist when it does.

Consider a differential file: a separate file of changes to a main database. Every night, the changes are incorporated into the database. Meanwhile, each database access must first check the differential file to see if the record of interest has been modified. A Bloom filter can reduce accesses to the differential file. For instance, each time a record is updated, you hash the record key with this technique. Then, you access a record check to see if there is a hit against the hash file. If there is no hit, you can be guaranteed that the record was not modified. If there is a hit, you must search the differential file.

How about a hyphenation routine with a general rule and a table of exceptions? If you don't find the word in the exception hash file, use the general rule. If there is a hit, search the word database for the particular exception.

Bloom filters can even work as spelling checkers. Occasionally a nonword "passes," but the dictionary can be stored in far less space than it would be as individual words.

In this month's column, Bill Stallings uses Bloom filters in a similar application. Instead of checking for correctly spelled words, however, he uses Bloom filters to check for easy-to-guess passwords like those that made the 1989 Internet Worm an infamous part of computer lore. As you might guess, such passwords, which are highly susceptible to computer breakins, bring smiles to crackers' faces, and Bill's approach to Bloom filters and computer-generated passwords should be seriously considered.

I look forward to hearing from you about the algorithms you find most useful, algorithms you'd like to find out more about, or those that you've developed and that you'd like to share with other *DDJ* readers. You can contact me at schneier@chinet.com, or through the *DDJ* offices.

A system intruder's objective is to gain access to your computer system or to increase the range of privileges accessible on it. Generally, this requires that the intruder acquire information that should have been protected, usually via user passwords. With knowledge of someone else's password, the intruder can log into a system and exercise all the privileges accorded to the legitimate user.

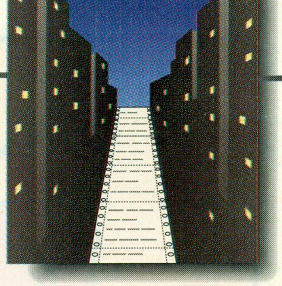
Left to their own devices, many users choose a password that is too short or too easy to guess. However, if users are assigned passwords consisting of, say, eight randomly selected, printable characters, password cracking can be effectively rendered impossible. The problem with this approach is that most users can't remember such passwords. Fortunately, even if we limit the password universe to strings of characters that are reasonably memorable, the size of the universe is still too large to permit practical password cracking. Our goal, then, should be to eliminate guessable passwords while allowing the user to select a password that is still memorable. There are four basic techniques currently in use to enable this:

- User education.
- Computer-generated passwords.
- Reactive password checking.
- Proactive password checking.

Users can be told the importance of using hard-to-guess passwords and can be provided with guidelines for selecting strong passwords. This user-education strategy is unlikely to succeed at most locations because many users will simply ignore the guidelines, while others may not be good judges of what is a strong password. For example, many users (mistakenly) believe that reversing a word or capitalizing the last letter makes a password unguessable.

Computer-generated passwords also have problems. If the passwords are random in nature, users will not be able to remember them. Even if the password is pronounceable, the user may have difficulty remembering it and so be tempted to write it down. In general, computer-generated password schemes have a history of poor acceptance by users.

William is president of Comp-Comm Consulting of Brewster, MA. He is the author of over a dozen books on data communications and computer networking, including Network and Internetwork Security *(Prentice-Hall, 1994). He can be reached at stallings@acm.org.*



Password Generation by Bloom Filters

William Stallings

Introduction

by Bruce Schneier

Niklaus Wirth said: "Algorithms+data structures=programs." Every computer program consists of complex algorithms: to sort the database, compute the formulas, draw the graphics, and display the data. Often, the difference between a good program and a bad one is the underlying algorithms.

"Algorithm Alley" explores the design and implementation of algorithms. Every month I'll present useful algorithms that you can implement today. The algorithms will cover a variety of areas — computation, graphics, databases, networking, artificial intelligence, and more — and be relevant to many more applications.

The ultimate goal of this column is to help you think about algorithms so you can develop your own. A craft so varied as programming cannot be taught as a series of recipes. No matter how many algorithms I present, you're going to need something else. If I can teach you general principles of algorithms, then you can take them with you wherever you program.

My first column is about Bloom filters, a method of hashing that greatly reduces memory requirements at the expense of false "hits." They are useful in a variety of applications, particularly those in which no calculation is required if the search is unsuccessful. For example, you might want to check someone's credit rating or passport number, but do nothing else if the record doesn't exist. While Bloom filters will occasionally report that a record exists when it doesn't, they'll never erroneously report that a record doesn't exist when it does.

Consider a differential file: a separate file of changes to a main database. Every night, the changes are incorporated into the database. Meanwhile, each database access must first check the differential file to see if the record of interest has been modified. A Bloom filter can reduce accesses to the differential file. For instance, each time a record is updated, you hash the record key with this technique. Then, you access a record check to see if there is a hit against the hash file. If there is no hit, you can be guaranteed that the record was not modified. If there is a hit, you must search the differential file.

How about a hyphenation routine with a general rule and a table of exceptions? If you don't find the word in the exception hash file, use the general rule. If there is a hit, search the word database for the particular exception.

Bloom filters can even work as spelling checkers. Occasionally a nonword "passes," but the dictionary can be stored in far less space than it would be as individual words.

In this month's column, Bill Stallings uses Bloom filters in a similar application. Instead of checking for correctly spelled words, however, he uses Bloom filters to check for easy-to-guess passwords like those that made the 1989 Internet Worm an infamous part of computer lore. As you might guess, such passwords, which are highly susceptible to computer break-ins, bring smiles to crackers' faces, and Bill's approach to Bloom filters and computer-generated passwords should be seriously considered.

I look forward to hearing from you about the algorithms you find most useful, algorithms you'd like to find out more about, or those that you've developed and that you'd like to share with other *DDJ* readers. You can contact me at schneier@chinet.com, or through the *DDJ* offices.

A system intruder's objective is to gain access to your computer system or to increase the range of privileges accessible on it. Generally, this requires that the intruder acquire information that should have been protected, usually via user passwords. With knowledge of someone else's password, the intruder can log into a system and exercise all the privileges accorded to the legitimate user.

Left to their own devices, many users choose a password that is too short or too easy to guess. However, if users are assigned passwords consisting of, say, eight randomly selected, printable characters, password cracking can be effectively rendered impossible. The problem with this approach is that most users can't remember such passwords. Fortunately, even if we limit the password universe to strings of characters that are reasonably memorable, the size of the universe is still too large to permit practical password cracking. Our goal, then, should be to eliminate guessable passwords while allowing the user to select a password that is still memorable. There are four basic techniques currently in use to enable this:

- User education.
- Computer-generated passwords.
- Reactive password checking.
- Proactive password checking.

Users can be told the importance of using hard-to-guess passwords and can be provided with guidelines for selecting strong passwords. This user-education strategy is unlikely to succeed at most locations because many users will simply ignore the guidelines, while others may not be good judges of what is a strong password. For example, many users (mistakenly) believe that reversing a word or capitalizing the last letter makes a password unguessable.

Computer-generated passwords also have problems. If the passwords are random in nature, users will not be able to remember them. Even if the password is pronounceable, the user may have difficulty remembering it and so be tempted to write it down. In general, computer-generated password schemes have a history of poor acceptance by users.

William is president of Comp-Comm Consulting of Brewster, MA. He is the author of over a dozen books on data communications and computer networking, including Network and Internetwork Security *(Prentice-Hall, 1994). He can be reached at stallings@acm.org.*

Reactive Password Checking

A reactive password-checking strategy is one in which the system periodically runs its own password cracker to find guessable passwords. The system cancels any passwords that are guessed and notifies the user. This tactic has a number of drawbacks. First, it is resource intensive if the job is done right. Because a determined opponent who is able to steal a password file can devote hours or even days of full CPU time to the task, an effective reactive password checker is at a distinct disadvantage. Furthermore, any existing passwords remain vulnerable until the reactive password checker finds them.

Proactive Password Checking

The most promising approach to improved password security is a proactive password checker. In this scheme, a user is allowed to select his or her own password. However, at the time of selection, the system checks to see if the password is allowable and, if not, rejects it. Such checkers are based on the philosophy that, with sufficient guidance from the system, users can select memorable passwords from a fairly large password space that are not likely to be guessed in a dictionary attack.

The trick with a proactive password checker is to strike a balance between user acceptability and password strength. If the system rejects too many passwords, users will complain that it is too hard to select a password. If the system uses a simple algorithm to define what is acceptable, password crackers, too, can refine their guessing technique.

The most straightforward approach is a simple system for rule enforcement. For example, all passwords have to be at least eight characters long, or the first eight characters must include at least one uppercase letter, one lowercase letter, one numeral, and one punctuation mark. These rules could be coupled with advice to the user. Although this approach is superior to simply educating users, it may not be sufficient to thwart password crackers. This scheme alerts crackers as to which passwords *not* to try, but may still make it possible to do password cracking.

Another possible procedure is simply to compile a large dictionary of possible "bad" passwords. When a user selects a password, the system checks to make sure that it is not on the disapproved list. However, one problem with this approach is space—the dictionary must be very large to be effective. Another problem is time—the time required to search a large dictionary may itself be great. In addition, to check for likely permutations of dictionary words, either those words must be included (making it truly huge), or each search must also involve considerable processing.

(continued from page 120)

Bloom Filters

A different approach is based on the use of Bloom filters, a hashing technique that makes it possible to determine, with high probability, whether a given word is in a dictionary. The amount of online storage required for the scheme is considerably less than that required to store an entire dictionary of words and permutations, and the processing time is minimal. Eugene Spafford and his colleagues at Purdue have adapted the Bloom filter for proactive password checking.

A Bloom filter of order k consists of a set of k independent hash functions $H_1(x)$, $H_2(x)$,..., $H_k(x)$, where each function maps a word into a hash value in the range 0 to $N-1$. That is, $H_i(X_j)=y$ $1\le i\le k$; $1\le j\le D$; $0\le y\le N-1$, where X_j=jth word in the password dictionary, and D=number of words in the password dictionary. This procedure is then applied to the dictionary: 1. A hash table of N bits is defined, with all bits initially set to 0; 2. for each dictionary word, its k hash values are calculated, and the corresponding bits in the hash table are set to 1. Thus, if $H_i(X_j)=67$ for some (i,j), then the 67th bit of the hash table is set to 1; if the bit already has the value 1, it remains at 1.

When a new password is presented to the checker, its k hash values are calculated (see Figure 1). If all the corresponding bits of the hash table are equal to 1, then the password is rejected. All passwords in the dictionary will be rejected. But there will also be some "false positives"—passwords that are not in the dictionary but that do produce a match in the hash table. To see this, consider a scheme with two hash functions. Suppose that the passwords *undertaker* and *hulkhogan* are in the dictionary, but *XlnsXqtn* is not. Further suppose that:

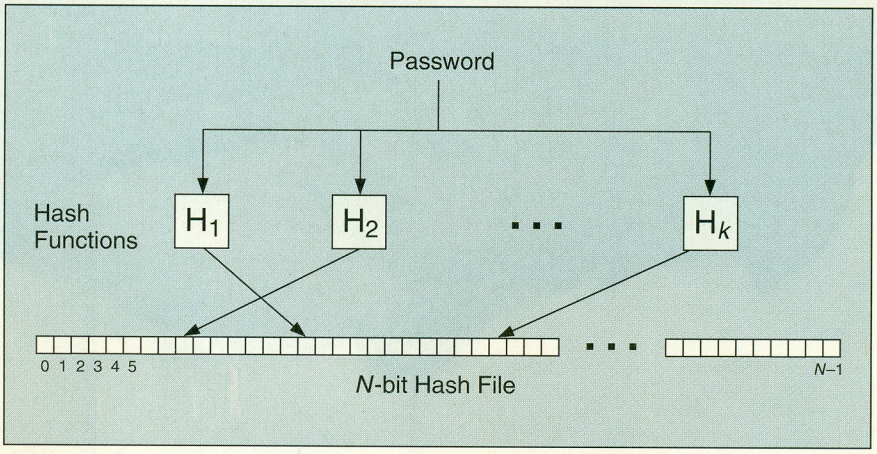


Figure 1: *Password checking with a Bloom filter.*

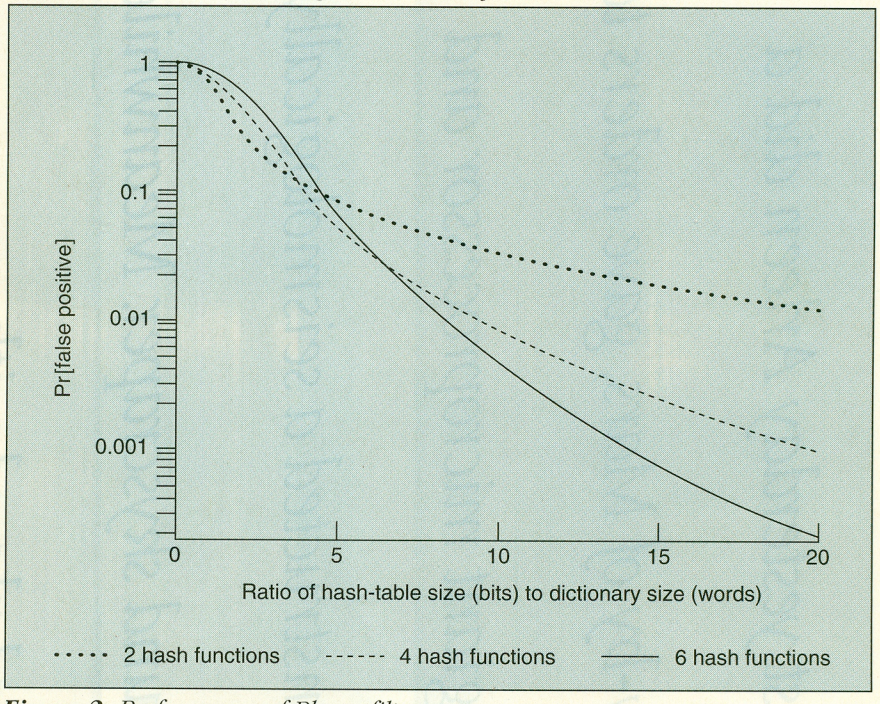


Figure 2: *Performance of Bloom filter.*

(a) $$P \approx \left(1-e^{-kD/N}\right)^k \approx \left(1-e^{-k/R}\right)^k$$

(b) $$R \approx \frac{-k}{\ln\left(1-P^{1/k}\right)}$$

Example 1: *Approximating the probability of a false positive.*

H_1(undertaker)=25
H_1(hulkhogan)=275
H_1(XlnsXqtn)=665
H_2(undertaker)=998
H_2(hulkhogan)=665
H_2(XlnsXqtn)=998

If the password *XlnsXqtn* is presented to the system, it will be rejected even though it is not in the dictionary. If there are too many such false positives, it will be difficult for users to select passwords. Therefore, you would like to design the hash scheme to minimize false positives. It can be shown that the probability of a false positive can be approximated by the equation in Example 1(a) or, equivalently, Example 1(b), where k=number of hash functions, N=number of bits in hash table, D=number of words in dictionary, and R=(N/D), the ratio of hash-table size (bits) to dictionary size (words).

Figure 2 plots P as a function of R for various values of k. Suppose you have a dictionary of one million words and wish to have a 0.01 probability of rejecting a password not in the dictionary. If you choose six hash functions, the required ratio is R=9.6. Therefore, you need a hash table of 9.6×10^6 bits, or about 1.2 megabytes of storage. In contrast, storage of the entire dictionary would require on the order of eight megabytes. Thus, you achieve a compression factor of almost 7. Furthermore, password checking involves the straightforward calculation of six hash functions and is independent of the size of the dictionary, whereas with the use of the full dictionary, there is substantial searching.

References

Bloom, B. "Space/time Trade-offs in Hash Coding with Allowable Errors." *Communications of the ACM* (July 1970).

Spafford, E. "Observing Reusable Password Choices." *Proceedings, UNIX Security Symposium III*, September 1992.

——. "OPUS: Preventing Weak Password Choices." *Computers and Security* (No. 3, 1992).

Stallings, W. *Network and Internetwork Security: Principles and Practice.* Englewood Cliffs, NJ: Prentice-Hall, 1994.

DDJ

To vote for your favorite article, circle inquiry no. 12.