

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

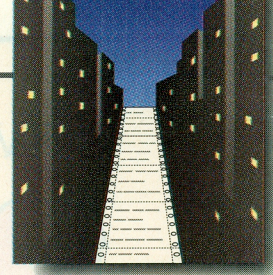
William is president of Comp-Comm Consulting of Brewster, MA. He is the author of over a dozen books on data communications and computer networking, including Network and Internetwork Security (Prentice-Hall, 1994). He can be reached at stallings@acm.org.

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.



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

William is president of Comp-Comm Consulting of Breuster, 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.

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.

Save Disk Space

PKZIP®

PKZIP version 2.0

PC WORLD

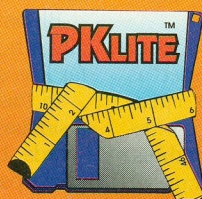
WORLD CLASS
AWARD

PKWARE® introduces the next generation of its award winning compression utility. PKZIP 2.0 yields greater performance levels than achieved with previous releases of the software. PKZIP compresses and archives files. This saves disk space and reduces file transfer time.

Software developers! You can significantly reduce product duplication costs by decreasing the number of disks required to distribute your applications. Call for Distribution License information.

Put Your Executables on a Diet

Software developers! Save disk space and media costs with smaller executables. You can distribute your software in a compressed form with PKLITE Professional™. PKLITE Professional gives you the ability to compress files so that they cannot be expanded by PKLITE™. This discourages reverse engineering of your programs.



PKLITE increases your valuable disk space by compressing DOS executable (.EXE and .COM) files by an average of 45%. The operation of PKLITE is transparent, all you will notice is more available disk space!

Compression for YOUR Application



The PKWARE Data Compression Library™ allows you to incorporate data compression technology into your software applications. The application program controls all the input and output of data, allowing data to be compressed or extracted to or from any device or area of memory.

All Purpose Data Compression Algorithm compresses ASCII or binary data quickly. The routines can be used with many popular DOS languages. A Windows DLL and an OS/2 32-bit version is also available!

PKWARE® INC.

The Data Compression Experts®

9025 N. Deerwood Drive Brown Deer, WI 53223-2437
(414) 354-8699 Fax (414) 354-8559

PKWARE Data Compression Library for DOS \$275 PKWARE Data Compression Library for OS/2 \$350
PKWARE Data Compression Library DLL for Windows \$350
PKZIP \$47.00 PKLITE \$46.00 PKLITE Professional \$146.00
Please add \$5.00 S&H per package in the US & Canada, \$11.25 overseas.
Wisconsin residents add appropriate state sales tax & county sales tax.
Visa and Mastercard accepted, no COD orders.

DD8-94

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.

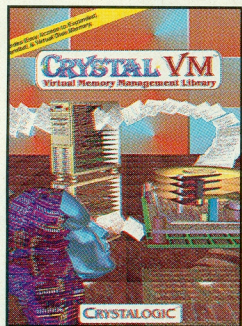
The trick with a proactive password checker is to strike a balance between user acceptability and password strength. If the system rejects too many passwords, users will complain that it is too hard to select a password. If the system uses a simple algorithm to define what is acceptable, password crackers,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

Virtual Memory Management Library

Now you can access all available memory without the complexity, learning curve and expense of a DOS extender.



The Crystallogic **VIRTUAL MEMORY MANAGEMENT LIBRARY** for C/C++ provides easy access to Expanded, Extended & Virtual Disk Memory. It offers instant access to all available memory from DOS based C/C++ programs. Easy to learn with "no royalties". Now your programs can dynamically determine and use all available memory resources. Perfect for graphics, data collection, large data files or any applications that use large amounts of data.

Includes support for Microsoft, Borland or any ANSI C or C++ compiler. No Royalties.

\$249.00

Visa, Mastercard & American Express Accepted

REQUIREMENTS:
IBM or Compatible PC, MS-DOS V 3.3 or later,
One 5.25 or 3.5" floppy disk drive, ANSI C/C++ compiler



CRYSTALOGIC™

PERFECTLY LOGICAL SOFTWARE
2525 Perimeter Place Dr., Suite 121 • Nashville, TN 37214

To Order Call 1-800-915-6442

FOR MORE INFORMATION 1-800-391-9190
FAX 615-391-5292 • BBS 615-391-8065
Dealer, OEM and VAR's inquires welcomed.

(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, $H_i(X_j) = y$ $1 \leq i \leq k$; $1 \leq j \leq D$; $0 \leq y \leq N-1$, where $X_j = j$ th word in the password dictionary, and $D = \text{number of words in the password dictionary}$. 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 *bulkbogan* are in the dictionary, but *XlmsXqtn* is not. Further suppose that:

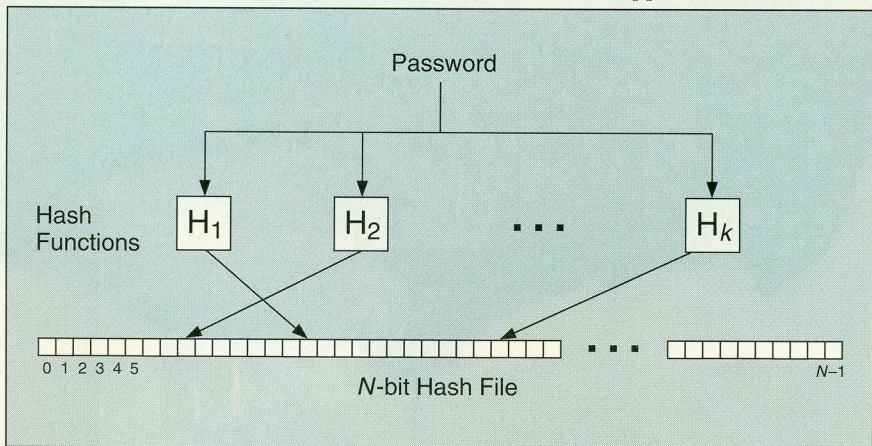


Figure 1: Password checking with a Bloom filter.

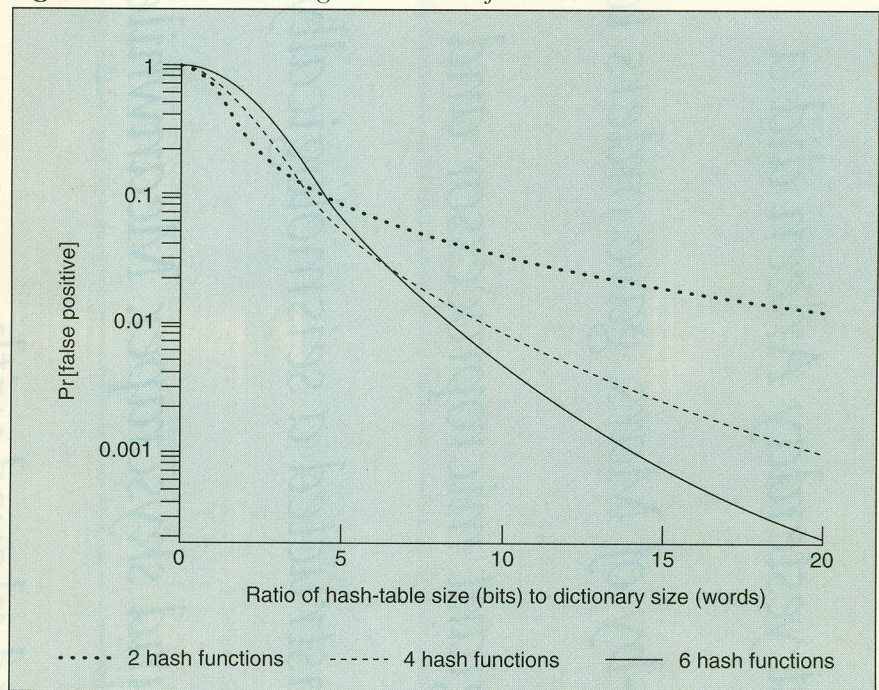


Figure 2: Performance of Bloom filter.

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

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

Example 1: Approximating the probability of a false positive.

$H_1(\text{undertaker})=25$
 $H_1(\text{hulkhogan})=275$
 $H_1(\text{XlnsXqtn})=665$
 $H_2(\text{undertaker})=998$
 $H_2(\text{hulkhogan})=665$
 $H_2(\text{XlnsXqtn})=998$

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).

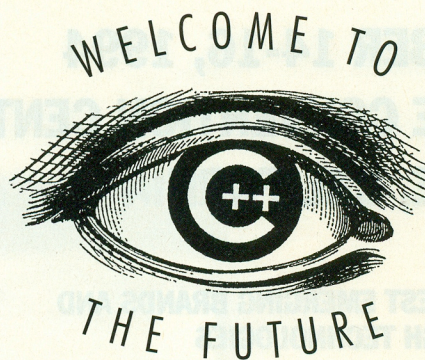
Spafford, E. "Observing Reusable Password Choices." *Proceedings, UNIX Security Symposium III*, September 1992.

—. "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.

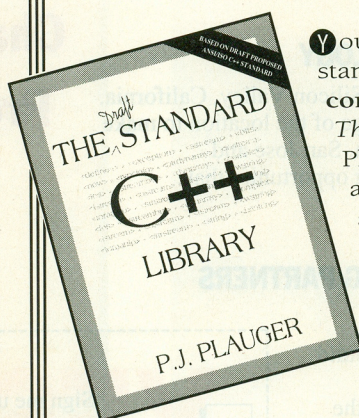


REUSABLE, PORTABLE CODE TAKES A MAJOR STEP FORWARD

NEW FROM PTR PRENTICE HALL...

If you're trying to write C++ code that really is reusable and portable, here's good news:

A standardized set of C++ libraries is on the way. Working with just about any C++ implementation, they give you powerful new tools for writing code that truly meets the promises of object technology.



You need to understand this new standard, and **there's only one comprehensive book** on the subject: *The Draft Standard C++ Library*, by P.J. Plauger. It covers every library class and function mandated by the standard. It includes a working implementation, with **thousands of lines of highly-portable sample code** you can start using today. It even includes the actual text of the proposed standard, with answers to detailed technical questions.

You'll learn about **pioneering features** such as library templates, exceptions and namespaces — **and gain practical insight into the C++ libraries you're already using.**

Nobody's better qualified to present this essential information than P.J. Plauger. He wrote the leading reference to C libraries, *The Standard C Library*, and has served as editor for the library portion of the draft C++ standard.

Whether you're adapting C++ class libraries or creating them, this book is invaluable.

You'll find it at your participating PTR Prentice Hall Magnet Store. To locate the nearest store, fax 201-816-4146, or ftp to ftp.prenhall.com. (Log in as *Anonymous* and use your e-mail address as your password.) Or gopher to gopher.prenhall.com.



Magnet Store

THE PROFESSIONAL'S CHOICE

ISBN:0-13-117003-1