

An Algorithm for Approximate Membership Checking With Application to Password Security

Udi Manber¹ and Sun Wu

Department of Computer Science
University of Arizona
Tucson, AZ 85721

June 1992

Revised versions December 1992 and January 1994

Appeared in *Information Processing Letters* **50** (1994) 191-197

Keywords: Approximate string matching, Bloom filters, data structures, design of algorithms, spell checking, password security.

Abstract

Given a large set of words W , we want to be able to determine quickly whether a query word q is close to any word in the set. A new data structure is presented that allows such queries to be answered very quickly even for huge sets if the words are not too long and the query is quite close. The major application is in limiting password guessing by verifying, before a password is approved, that the password is not too close to a dictionary word. Other applications include spelling correction of bibliographic files and approximate matching.

1. Introduction

People have tried to guess passwords for a long time in many different situations (see for example, [MT79] and [K190]). Many computer systems have been compromised because of weak passwords. Distribution of words that may serve as passwords (and thus should be avoided) has even reached the popular press [Ha91]. It's an unusual race. Hackers try to find new emerging words that may give someone ideas for passwords, and administrators (and increasingly ordinary users who are worried) try to add these words as soon as possible to their list of forbidden passwords. There have been suggestions of a national bank — sort of a super dictionary — that will be updated regularly and distributed through the Internet [Sp92]. With the increased efficiency of encrypting, and thus guessing, it is now possible to guess millions of words in a reasonably short

¹ Supported in part by an NSF Presidential Young Investigator Award (grant DCR-8451397), with matching funds from AT&T, and by an NSF grant CCR-9001619.

time. This is coupled with the nonchalant way most people regard their passwords, and the ignorance of most users to the risks involved in a compromised password. In a test done at the Computer Science department at the University of Arizona (with mostly non-naive users) a while ago 229 passwords were successfully guessed [To91]. Following that test, several new guidelines were introduced that restrict passwords in what seemed like a secure way. Nevertheless, in a simple test, one of the authors was able to guess two passwords overnight, both of which adhered to the guidelines.²

There are essentially two approaches to the problem. The first is to assign users random passwords. That may seem secure, but the problem is that people cannot remember a random password and they proceed to write it down in an obvious place, like the bottom drawer. Or they don't write it down, then forget, and then lose either data or time (see [MT79] for other weaknesses of this approach). The second approach is to allow users to select their own passwords, but enforce strict rules about them. (This is often done right after the system has been compromised.) There is a popular and effective rule that seems to prevent password guessing: do not accept a password that consists of only alphanumeric characters. By forcing the users to include at least one non-alphanumeric character, such as '.', ';', ',', it is almost guaranteed that the password will not appear in any dictionary and thus cannot be guessed. Some extra precautions are usually taken, for example, the special character must not appear at the beginning or the end (so that 'yes;' or '.no' are not acceptable). Restricting the space of allowable passwords makes it harder for the user, but it appears necessary, as we discuss below, because many more passwords can now be guessed in a short time. Surprisingly the idea of enforcing the use of non-alphanumeric characters (and checking the passwords only against a relatively small dictionary containing only phrases with non-alphanumeric characters) is not mentioned in [Sp92], which concentrates entirely on checking passwords against a large dictionary. This idea is quite common. It is easier to verify that a password contains a special character than it is to check the password against a large dictionary; it is easier for the users to generate a password they can remember (e.g., pass.word or gue;ss); and it gives a misleading sense of security.

We argue, however, that this is not secure enough. Given a reasonable size dictionary of, say 100,000 words, a guesser can generate all possible combinations of these words with one additional non-alphanumeric character. Suppose that we try 10 different non-alphanumeric characters (which covers all popular ones), and that we insert them in all positions and also substitute any position in the word with all those characters. If the average length of a word is 7 characters, there are 7 places to substitute, and 6 places to insert (excluding the beginning and end). Overall, we have a total of approximately $100,000 * 10 * (7+6) = 13,000,000$, a large number, but not prohibitively large. It is possible to guess that many combinations in a reasonable time. For example the password *Ae-ean* may seem random enough and it will not likely appear in any dictionary but

² These were real tests, not attempts to gain unauthorized access. All users whose passwords were compromised were immediately notified and asked to change them.

it happens to be within distance 1 of *Aegean* which is likely to appear in every dictionary, and thus may be one of the guesses. In addition, some passwords that contain one non-alphanumeric character may still belong to some dictionaries of common words; for example, ‘In/Out,’ ‘a.out,’ or ‘Los-Angeles’. We need to verify that the password is *not too close* to “known words.”

We suggest that, in addition to other rules, passwords should not be within distance 1 (i.e., insertion, deletion, or substitution) to a word in a dictionary.³ When a new password is entered, it should be checked against the dictionary. One can use a program for approximate string matching, such as *agrep* [WM92], but this involves reading the whole dictionary. In this paper, we present a simple data structure that allows very fast approximate queries provided that the distance is small, and in particular 1. The data structure is based on the idea of Bloom filters together with additional features. Although preventing password guessing is the primary focus in this paper, the algorithms can also be used in other applications requiring fast approximate queries to large databases. We mention two of them here.

The first application is in checking spelling in large bibliographic files. The problem with such files is that they contain a large number of names, which will be flagged as errors. One needs to look at the list by hand to find real misspellings. Our idea is to first use the regular spell checker and find the list of words in the file that are not in the dictionary. Then use our scheme to find, among these words, those that are within distance 1 to a word in a dictionary. Any misspelling involving one character will still be found, but many names will be eliminated. We are currently experimenting with and improving this scheme. In a preliminary test on a large bibliographic file (a list of most of the papers published in the theory of computing area, maintained by Joel Seiferas, containing ~30,000 entries and occupying ~3MB), the UNIX *spell* program found 11,251 "misspellings." After some basic filtering, we got this number down to 8305. Then we used our algorithm, and found that only 705 (about 8%) of those words are within a distance of 1 to the dictionary. They still included mostly names, special terms, and foreign words (some of the entries were in French and German), but we did find 45 misspelled words. Looking at 705 words is much easier than looking at 8305 words.

The second application is to the general problem of approximate matching. A very powerful tool in approximate matching is *filtering*. The idea is to relax a given pattern-matching problem to an easier problem whose output is a superset of the output of the original problem. The easier problem is used as a filtering device, hopefully pruning the input substantially. The original problem is then solved only on the filtered set. For example [WM92], if the pattern is a string of length p , and there is an approximate match with k errors, then there is an exact match to at least one substring of length $r = \lfloor p/k + 1 \rfloor$ of the pattern. We can therefore cut the pattern into pieces of smaller size and look for exact matches to any one of these pieces. An exact match to one piece

³ Which dictionary to use depends on the desired security. Hopefully, very large sets of ‘common words’ will be available in the public domain.

does not imply an approximate match to the whole pattern, but we can filter the input substantially. There are other ways to reduce an approximate-matching problem to a different exact-matching problem. It is also possible to reduce a k -error matching problem to smaller 1-error matching problems, and having a fast algorithm for the 1-error matching problem is thus important. There will be fewer pieces than in the reduction to the exact case and the pieces will be larger. This approach is not always beneficial. As in most other filtering techniques, it depends heavily on the properties of the patterns and the exact parameters of the search. We do not have a specific application for which we can show that a reduction to 1-error matching is better than all other techniques, but we believe that it deserves further study. This general idea is also discussed in [PW93].

2. The Data Structure and the Algorithm

We solve the approximate membership problem by extending the original Bloom filters in two ways. The first extension is a reduction of the approximate problem to several instances (but not too many) of the exact problem. The second extension is an improvement in the query time by utilizing locality.

First, we describe the original Bloom filter [B170], which is the basis of our data structure. Let $W = w_1, w_2, \dots, w_n$ be a set of words. Let h_1, h_2, \dots, h_k be k independent hash functions that map words into a hash table of size M . For each word $w_i \in W$, we mark the locations $h_1(w_i), h_2(w_i), \dots, h_k(w_i)$ in the hash table (if a location is already marked we do nothing). (In [B170], the assumption was that the k locations are chosen without repetitions; it is also possible to allow repetitions, which makes the program simpler.) A marked bit indicates that *at least* one word has been hashed to it. Given a query word q , we hash q in the same way using all k hash functions. If q belongs to W , all corresponding locations would be marked; therefore, if any of those locations is unmarked, then q clearly does not belong to W . There is a chance that all locations are marked and q still does not belong to W . The likelihood of this happening depends on the values of n , k , and M . By setting appropriate values, we can make the probability of error reasonably small (although not zero). The only errors are *false positive* errors. Members of the set will always be identified as such. Bloom filters are useful for cases where a small number of false positive errors can be tolerated. They are used, for example, in the UNIX spell program to check words against the dictionary. (It is interesting to note that the default values set by the spell program are such that the probability of error is about 0.5%, which is quite high for spell checking; those numbers were set more than 15 years ago when disk space was at a premium.) In [Sp92], Bloom filters were suggested as the basis for checking all passwords against the dictionary.

Bloom filters, and indeed any method that relies entirely on hashing, are very good for exact queries, but they cannot be used for approximate queries, because a small change in the input makes an arbitrary change in the hash values. We extend the idea in the following way. Let $q = x_1x_2 \cdots x_d$ be a query word, where the x_i 's are characters from a fixed finite alphabet (e.g., ASCII). We define the word $q-i$, for an integer $1 \leq i \leq d$, such that $q-i =$

$x_1x_2 \cdots x_{i-1}x_{i+1} \cdots x_d$. In other words, $q-i$ is obtained by removing the i 'th character from the word q . We define an *extended word* (q,j) to be a word followed by an integer. The meaning of an extended word (q, j) is to add a 'don't-care' symbol after position j in q . Given a word q , we form a set of extended words, called the *1-extension* of q , and denoted by Q^1 , as follows:

$$Q^1 = \{ (q, 0), (q, 1), \dots, (q, d), (q-1, 0), (q-2, 1), \dots, (q-d, d-1) \}.$$

Q^1 contains $2d+1$ extended words. The first $d+1$ extended words correspond to insertions (at positions 0 to d). The last d extended words correspond to substitutions (at position 1 to d). Let W^1 be the union of all sets of extended words formed from the words in W . The next theorem forms the heart of our technique.

Theorem 1: There is a word w in W that is within a distance of 1 from q if and only if there is an extended word in W^1 that equals (exactly) to an extended word in Q^1 .

Proof: Suppose that there is an extended word in W^1 , say (w', i) , which is formed from the word w , such that $(w', i) = (q', i)$ formed from q . There are four cases depending on whether or not $w = w'$ and $q = q'$. If both $w = w'$ and $q = q'$, then clearly $w = q$ and we are done. If $w = w'$, but $q \neq q'$, then $q' = q - (i+1)$, and a deletion to q at position $i+1$ will make it equal to w . The case of $w \neq w'$ and $q = q'$ is similar except that the deletion is to w (which is the same as an insertion to q). If $w \neq w'$ and $q \neq q'$, then w and q differ at exactly the same location, which implies that a substitution at that location will make them equal.

We also have to prove the opposite; namely, if there is a word w in W that is within distance 1 of q , then we can find $(q', i) = (w', i)$. There are 3 cases, depending on whether the distance is caused by an insertion, a deletion, or a substitution. If w can be obtained from q by an insertion after position i , then $q' = q$ and $w' = w - i$; if w can be obtained from q by a deletion of character i , then $w' = w$ and $q' = q - i$; and if we need a substitution at position i , then $q' = q - i$ and $w' = w - i$. \square

Theorem 1 reduces the problem of approximate queries of distance 1 to exact queries from a larger set. It is possible, of course, to extend the theorem to cover a distance of 2, for example, but that requires $O(k^2)$ queries per word. It is also possible to allow other types of errors. For example, we may want to consider the removal of, say, two '*'s (or any other character) in a row, to be one deletion.

Given a set W , we construct W^1 and hash all its members, using the Bloom filter technique, to a large hash table. We never need to actually store the extended words, only the hash table. The Bloom filter technique has another advantage for us. If the input W^1 is a multiset — that is, some elements may appear several times — the different copies of the same element will be hashed to the same places and will have no effect on the size of the hash table. Therefore, there is no need to sort and remove duplicates, because it will be done implicitly. Also, it is easy to add words to the set at any time.

Given a query q , to find whether it is within distance 1 to W , we construct Q^1 and check all its extended words for membership in W^1 . We therefore have $2d+1$ exact queries to perform,

which poses the following problem. If the dictionary is large, which is the case here, the hash table will reside on a disk. Each access to the hash table will likely lead to a different page. Each exact query requires k hash table lookups, and we have $2d+1$ exact queries. The number of pages that we need to fetch is therefore approximately $k(2d+1)$, which is too high, and may outweigh all other costs. Fortunately, the basic idea of Bloom filters can be improved to minimize the number of page fetches.

Instead of allowing the k hash functions to mark k arbitrary locations in the hash table, we will insist that all k locations *are in the same page*. We do that in a two-step approach. First, we hash each element to a particular page, using an extra hash function that maps to the range of 1 to m/p , where m is the size of the global hash table, and p is the size of a page. Then, we consider that page to be the whole hash table and use the k hash functions to map the element into it. The problem is that the probability for false positives for this scheme is larger than the probability for the original scheme, because the hash values are more clustered. The analysis turns out to be much more complicated (see next section). Fortunately, for the range of values we are interested in — a large hash table and page sizes of more than 1000 bits — the differences in the probabilities are small, rendering this variation of Bloom filters very attractive for secondary storage. (It becomes even more attractive for CD storage where page access is slower.) Of course, for only one use of the algorithm (e.g., to check one password) speed is not a big issue. But our algorithm may be used many times (for other applications), and the differences in error rates are negligible enough for us to recommend using this scheme all the time. Next, we study the time and space complexities of this approach, and then show empirically that both are good.

3. Analysis and Empirical Results

Analyzing the reduction from approximate queries to exact queries is trivial: there are $2k+1$ exact queries for each approximate query. The problem is the analysis of the error as a function of the hash table size and the number of hash functions used. This analysis is different from the one for the original Bloom filters because of the use of pages.

Denote the size of the hash table by M , the number of elements in W^1 by n , the page size (in bits) by p , and the number of hash functions that are used for each element by k . We will allow repetitions in this analysis to simplify it somewhat. We are interested in the probability that a random element, which does not belong to the set, will generate a false positive. (For each candidate word of size k , we actually check $2k+1$ words, but for now we will concentrate on the error rate for one check.) Let's denote this error probability function by $FP(M, n, p, k)$ (for False Positive). We will assume that the hash functions map the element into random uniform locations. Since each element is mapped into exactly one page, the question is to evaluate the probability that k random locations in one page are occupied after mapping n elements into random pages.

We begin by evaluating $FP(p, n, p, k)$, the probability of false positive for a hash table consisting on one page. Since there are n elements, each mapped by k hash functions, we hash $r \cdot k$

times into p locations with repetitions. Let $OC(p, n \cdot k, j)$ be the probability that exactly j of the p locations in the page are occupied. Computing $OC(p, n \cdot k, j)$ is a standard occupancy problem (see, for example, [Fe68], p. 102):

$$OC(p, n \cdot k, j) = \binom{p}{j} \sum_{i=0}^j (-1)^i \binom{j}{i} \left(\frac{j-i}{n}\right)^{nk}. \quad (3.1)$$

Assuming that exactly j cells are occupied, the probability that a query word leads to a false positive is equal to the probability of k random selections (again with repetition) all occupied. This probability is $(\frac{j}{p})^k$. Therefore, we obtain

$$FP(p, n, p, k) = \sum_{j=1}^p OC(p, n \cdot k, j) \cdot \left(\frac{j}{p}\right)^k. \quad (3.2)$$

Since each page is selected at random for each element, the probability of exactly r elements out of n mapped into a given page follows a Bernoulli distribution, which leads to the final result.

$$FP(M, n, p, k) = \sum_{r=0}^n \binom{n}{r} \left(\frac{p}{M}\right)^r \left(1 - \frac{p}{M}\right)^{n-r} \sum_{j=1}^p OC(p, r \cdot k, j) \cdot \left(\frac{j}{p}\right)^k. \quad (3.3)$$

The expression in (3.3) look formidable, and the problem of simplifying it is beyond the scope of this paper. In practice, however, our experiments show that $p \geq 1024$ is large enough to make the error probability of this scheme just about the same as the original Bloom filter (without repetitions), which is easy to analyze [BI70]. The probability that one location out of M is not occupied after k random locations are occupied is $1 - k/M$. Therefore, the probability that that location is not occupied after n elements are hashed is $(1 - \frac{k}{M})^n$, and the probability that k random locations are all occupied is

$$\left(1 - \left(1 - \frac{k}{M}\right)^n\right)^k \sim \left(1 - e^{-kn/M}\right)^k. \quad (3.4)$$

If n and M are given, then the value for k that minimizes (3.4) is $k = M/\ln 2 \cdot n = M/1.44n$. However, the larger k is the more work is involved, so k may be chosen to be less than $M/1.44n$. Since (3.4) is easy to compute, one can figure out the effects of smaller k and choose the best compromise. Figure 1 shows the error probability obtained from experiments for different values of p . The straight line with ■ is the value (0.94%) obtained from (3.4). These values were obtained for $n = 100,000$, $M = 1,000,000$, and $k = 5$. We chose $k=5$ instead of $k=7$ which minimizes (3.4) because the difference in the probabilities were small enough (it is 0.82% with $k=7$). Already for $p = 1024$ (which corresponds to 128 bytes) the difference is insignificant and can be attributed to the variance in the simulations. (All experiments were done by filling the table with n random elements, and then trying to check 1000 random elements for membership. Each experiment was repeated 50 times and the average is shown; standard deviation was around 30%.) Our recommendation is therefore to use (3.4) to determine the best compromise as long as p is not too small (and p is rarely if ever less than 32 bytes).

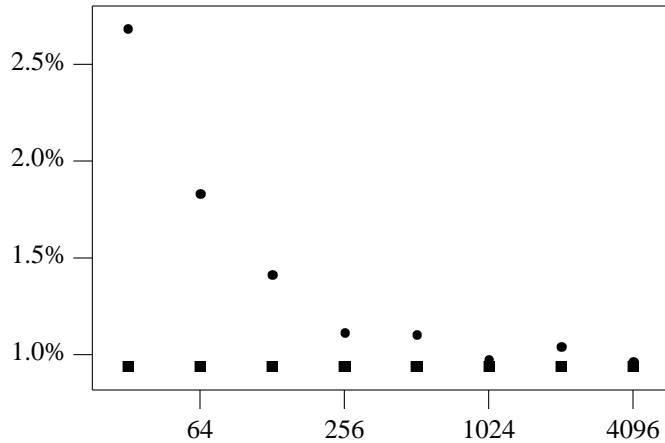


Figure 1: Error probabilities for the original (■) and modified (●) Bloom Filters.

To further verify the scheme, we tested the algorithms on real dictionaries. We used a large dictionary of size 1.6 MBytes, containing 194168 words. We then tested 10,000 random words (with at least one non-alphanumeric character to ensure that the word is not in the dictionary) of length 8, counted the rate of (false) acceptance and measured the average running time. We used 5 hash functions, and two hash table sizes, 5 Mbytes and 7.5 MBytes. We tested both the original Bloom filters and our localized scheme. Our scheme runs about 2.5 times faster and exhibits about the same error rate. Since we could not control the paging environment (nor the cache), we could not measure the precise effect of the local scheme. We used the user time and system time as given by UNIX, which is not very precise, but, since it is averaged over 10,000 words, can give a rough measure. The averages of the running times and error rates are given in Table 2.

To find the expected error rate implied by (3.4), we must take into account that each word is tested many times (according to its 1-extension); in our case, the 1-extension of each word contains $2 \cdot 8 = 17$ extended words. The 1-extension of the whole set (i.e., the set of all extended words obtained from the words in the dictionary) contained 3454916 words, but with many duplicates. For the purpose of the analysis, we checked the number of unique 1-extension words, which was 3026758. The expression in (3.4) shows the probability that one test comes out positive by mistake. The probability that any one of them is false positive is approximately

$$1 - (1 - (1 - e^{-kn/M})^k)^d. \quad (3.5)$$

Table 2 gives the theoretical error rate from (3.5), which matches the experiment very well.

Hash table size 7.5 MBytes				
	User time	System time	Error% experimental	Error% from (3.5)
Original	0.0082	0.161	0.94	0.93
Local	0.0051	0.062	0.98	
Hash table size 5 MBytes				
	User time	System time	Error% experimental	Error% from (3.5)
Original	0.0072	0.143	5.13	5.14
Local	0.0048	0.056	5.51	

Table 2: Running times and error rate for the password application.

4. Conclusions

We presented two extensions to Bloom filters to allow fast approximate set-membership tests. The first one was to reduce an approximate query to several exact queries and the second one was to utilize secondary memory more effectively by requiring that all hashing of the same element be directed to the same page. Together, these two extensions form a good data structure to check for weak passwords. They can also serve other application requiring fast approximate membership testing.

Acknowledgements

We thank Peter Downey and Gregg Townsend for helpful conversations. Cliff Hathaway and Gregg Townsend helped with the implementation.

References

- [BI70] Bloom, B. H., “Space/time trade-offs in hash coding with allowable errors,” *Comm. of the ACM*, **13** (July 1970), pp. 422–426.
- [Fe68] Feller, W., *An Introduction to Probability Theory and Its Applications*, Volume 1, third edition, John Wiley, 1968.
- [Ha91] *Icons of the computer age*, Harper’s Magazine, November 1991, p. 26.
- [K190] Klein, D. V., “A Survey of, and Improvement to, Password Security,” in *UNIX Security*

Workshop II, the Usenix Association (August 1990), pp. 5–14.

[MT79]

Morris, R. and K. Thompson, “Password security: a case history,” *Communications of the ACM*, **22** (November 1979), pp. 594–597.

[PW93]

Pevzner P. A., and M. S. Waterman, “A Fast Filtration Algorithm for the Substring Matching Problem,” in *4th Annual Combinatorial Pattern Matching Conf.* Padova, Italy, Springer-Verlag Lecture Notes in Computer Science #684 (June 1993), pp. 197–214.

[Sp92]

Spafford, E. H., “Opus: Preventing Weak Password Choices,” *Computers & Security*, **11** (May 1992), pp. 273–278.

[To91]

Townsend G., private communication, 1991.

[WM92]

Wu S., and U. Manber, “Fast Text Searching Allowing Errors,” *Communications of the ACM* **35** (October 1992), pp. 83–91.