High Dictionary Compression for Proactive Password Checking

FRANCESCO BERGADANO, BRUNO CRISPO, and GIANCARLO RUFFO
Università di Torino

The important problem of user password selection is addressed and a new proactive password-checking technique is presented. In a training phase, a decision tree is generated based on a given dictionary of weak passwords. Then, the decision tree is used to determine whether a user password should be accepted. Experimental results described here show that the method leads to very high dictionary compression (up to 1000 to 1) with low error rates (of the order of 1%). A prototype implementation, called ProCheck, is made available online. We survey previous approaches to proactive password checking, and provide an in-depth comparison.

Categories and Subject Descriptors: D.4.6 [Operating Systems]: Security and Protection—Access controls; Authentication
General Terms: Security
Additional Key Words and Phrases: Access control, decision trees, password selection, proactive password checking

1. INTRODUCTION

The problem of selecting and using good passwords is becoming more important every day. The number and the importance of services that are provided through computers and networks increases dramatically, and in many cases such services require passwords or other forms of user identification. For different reasons, including obvious security concerns, users have to use different passwords for different systems or services, making it more difficult to remember and protect one’s password. Passwords are not only critical for login identification, but also in more sophisticated service-granting systems, such as Kerberos [Neuman and Tso 1994]: in version 4,
an attacker can easily mount a password-guessing assault. He simply
records login dialogs, where answers are returned encrypted with a key
derived by a publicly-known algorithm from the user's password. A guess at
the user's password can be confirmed by calculating that key and using it to
decrypt the recorded answer. An attacker who has recorded many such
login dialogs has a high probability of finding new passwords because users
do not usually pick good ones unless forced to. Version 5, by introducing
preauthentication, makes the above attack harder, but the preauthentica-
tion mode is an option that the security manager can turn off, leaving the
system as vulnerable as in version 4. Moreover, if a third-party authentica-
tion sponsor is not used, the password-guessing attack is still possible if the
attacker can passively wiretap the network. On the other hand, the use of a
third party is rather expensive. Finally, if Kerberos is used by both clients
A and B to authenticate each other, instead of authenticating a client to a
server with preauthenticated data, the password-guessing attack is still
possible using the redundancy contained in the ticket given by the ticket-
granting service to client B and encrypted with the key derived from B's
password.

Finally, passwords are needed to protect secret information that cannot
be remembered by the user (e.g., private keys) in authentication and
encryption software essential to many applications.

The number of possible passwords is usually too large for brute-force
guessing because it is equal to \( \text{keyspace}^{\text{password length}} \). However, the number
of passwords that are likely to be selected in practice is much smaller. User
passwords often consist of simple variations on common words and rarely
include more than one special character. In cases more numerous than one
can imagine, the password is equal to the user name, or to other user-
related information that is easily obtained; see Klein [1990] for a case
study.

In multiuser environments, and most notably in Unix systems, password
guessing is one of the easiest and most common ways of attack. Excellent
public domain software exists to this purpose, such as the well-known
\texttt{crack} program [Muffett 1991]. The use of such attack techniques is made
easier in systems where the encrypted passwords are world-readable, as in
most Unix systems. More precisely, if an attacker can read the Unix
/etc/passwd file, he can then run programs such as \texttt{crack} to uncover
some user passwords, and then masquerade under the corresponding users
to perform more sophisticated attacks. A password-guessing program can
use this file for an offline attack. A truly huge number of passwords can be
tried for each user, and if the correct password is ever encountered, it is
detected. \texttt{Crack} is very effective in practice [Klein 1990]. Some versions of
\texttt{Crack}, like \texttt{QCrack} [ftp from dostoevsky.ucr.edu], achieve even greater
efficiency by performing password encryption offline, and storing the Unix
\texttt{crypt} output in a hash table. However, large amounts of disk space are
required, as Unix password encryption depends on a so-called “salt” param-
eter that may have different values.
On newer Unix systems, or by installing appropriate patches [Haugh 1991], password information can be moved to a so-called shadow file that is only readable by the superuser. This is definitely an improvement over previous Unix environments, as it disallows simple offline password guessing. However, this is far from a complete solution to the specific problem. First, users may use the same passwords on different systems, and not all systems may protect the file with the encrypted passwords. Second, the shadow password file might become readable for some reason (e.g., it is found on a stolen backup tape), resulting in the disclosure of some user passwords. Finally, even if an attacker has gained complete control of the system, he might need passwords belonging to normal existing users so as to connect them to false identities, and maybe prepare break-ins on other network sites.

Moreover, we may take a more general view, not restricted to particular implementations of Unix systems. The most common way of authenticating users is by means of passwords, and systems must have means to validate passwords. Password-related information is stored in memory, disks, and tapes, communicated through networks, shared among users in safe and less safe ways. In such scenarios, hard to guess passwords are essential.

Three techniques are generally known for selecting good passwords:

- **User education**—users are instructed on how to select good passwords. In general, this strategy does not work well because users do not follow the guidelines, especially in open environments where break-ins are more frequent.

- **Password generators**—the system chooses a random password for each user. From a theoretical point of view, this is the perfect solution. In practice, however, users tend to forget passwords, or worse, write them down or store them.

- **Password checking**—the passwords chosen by users are checked for possible weaknesses, and in this case the user is asked to select a different password. Two different approaches exist:
  - Reactive password checking: Programs such as crack are run periodically by the system administrators to find weak passwords and users of weak passwords are invited to change their passwords. There are at least three problems here: (1) users may nevertheless not change their passwords, or they may do so after considerable delay; (2) successful attacks may occur before the administrators run the check; (3) attackers may devote more cpu time and larger dictionaries to the password-cracking task. An implementation based on these principles is found in the COPS package [Farmer and Spafford 1990] or in the SPM package [Cooper 1995].

Typically, open environments are distributed systems where networks of dozens or even hundreds of computers are physically distributed in several, more or less secure, rooms or buildings and where many users—sometimes belonging to different organizational units—share access to the same machines.
Proactive password checking: When a user selects a password, the system checks immediately to see whether it is acceptable. If the password is weak (i.e., belongs to a dictionary of "easy" passwords), it is not accepted, and the user is asked to choose a new one. A disadvantage is the space required by dictionaries and the time to check. Time is important because the task is done online, while the user waits. Space becomes less important as storage devices become cheaper. However, cost reduction allows for more effective forms of attack, and larger dictionaries become necessary.

Proactive password checking is considered the best approach [Klein 1990; Bishop 1990; Spafford 1992], as it does not suffer from the difficulties of the other techniques. The literature on this subject is surprisingly limited, if one considers its great practical importance. Moreover, the proposed methods of proactive password checking are open to criticism and their performance is limited in either efficiency or reliability. In the next section, we shall review the best-known approaches. Our method\(^2\), called ProCheck, is described in Section 3, with experiments in Sections 4 and 5.

2. PROACTIVE PASSWORD CHECKING

In Unix systems, proactive password checkers usually come as patches or substitutes for password-changing programs such as passwd. A survey of proactive password checkers is found in Bishop [1992]. One implementation example is npasswd (available via anonymous ftp from emx.utexas.edu), and checks for minimum password length, host and user related information, repeated letters, and other simple password schemes, and membership in a few small dictionaries. All such tests are important, but the heart of a proactive password checker has to deal with detecting membership in large dictionaries. If the password belongs to one such dictionary, it is rejected.

There are two problems in this simple approach: space required to store the dictionaries, and time required to detect membership. Time is important because the user has to wait for the command prompt while the password is being checked. However, this occurs only occasionally, when a new password is selected. For a serious check, a dictionary of at least 30 Mbyte is preferable, but larger ones, including several languages, are needed in open sites with many users.

Previous research has mainly addressed the above space and time efficiency problems by substituting the direct membership test with alternative procedures. Our approach is also of this kind. We survey the simple, but effective, test proposed by Nagle [1988], the related BApasswd system [Davies and Ganesan 1993], and the approach based on Blum filters, described in Spafford [1992].

\(^2\)A short description of the method with some initial experiments is found in Bergadano et al. [1997]
Nagle [1988] proposes the following strategy. For each word in the dictionaries, scan the consecutive sets of three letters (trigraphs); register in a table that such trigraphs have occurred; then, when a possible password is presented, accept it only if at least two of its trigraphs have not occurred. As a consequence, words that are in the dictionary are all rejected because all of their trigraphs have been recorded. We refer to this by saying that the “false negative” error is zero. There will, however, be “false positives,” i.e., words that are not in the dictionaries, but are nevertheless rejected by the trigraph test. These are words such that all their trigraphs but one have occurred somewhere in the dictionary. When such errors occur, the user has to change her password, although it does not belong to the dictionary and seems to be acceptable. We also speak of a one-sided error. The error for false positives seems to be relatively low in practice, and the method is certainly attractive because of its simplicity. The space needed for recording a trigraph occurrence is of the order \(\text{keyspace}^3\). The complexity can be much lower in practice, as not all trigraphs occur in dictionaries. The limited size of the checker is a desirable aspect, but, as we argue later, is responsible for an error rate that is higher than in other methods. A 5% error rate of false positives has been observed in random passwords [Davies and Ganesan 1993].

A related approach, at least with respect to the use of trigraphs, is found in the BApasword system [Davies and Ganesan 1993]. There, a Markov model is obtained from the dictionary and is then used to obtain membership probabilities.

In BApasword, once the system is trained, a proposed password is given in input to the Markov model, and the overall password probability is computed by summing up the logarithms of the transition probabilities. This value is then normalized so that the mean is 0 and the standard deviation is 1. The password is then accepted if it lies outside a boundary of computed probability set to 2.6 times the standard deviation.

The general comments to be made for BApasword are not significantly different from the ones made for Nagle’s trigraph occurrence method. However, the Markov model has a two-sided error, as it can also classify as negative some words that actually belong to the dictionary. The method is relatively reliable and very attractive due to its small space requirement (a constant 175 Kbyte for a second-order model). However, this causes an error rate that is considerably higher than the one for Blum filters (described next) and is observed for the decision tree approach proposed in this paper. An in-depth comparison will follow the description of our experiments.

The method based on Blum filters and described in Spafford [1992] lies at the other end of the spectrum, in the sense that it does not achieve considerable dictionary compression, but has a low error rate. The procedure can be summarized as follows: a hash table of \(m\) bits is used, together with \(k\) independent hash functions; hash functions apply to dictionary words and produce integers in the range 1 to \(2^m\). In the training phase,
each word in the dictionaries is given in input to each of the \( k \) hash functions, and \( k \) integers are obtained. The bits for the binary representation of each of the integers are then set in the hash table. When checking a password, it is hashed with the \( k \) functions, and is accepted if at least one of the obtained bits is not set.

As it is the case for the trigraph test, and for similar reasons, all words in the dictionaries are rejected. Therefore, the probability of producing false negatives is 0, and we have a one-sided error. There can be, in fact, false positives, i.e., words not in the dictionaries that are rejected. The probability of such errors can be evaluated to \((1 - (1 - k/m)^n)^k\), where \( n \) is the number of words in the dictionaries. Let us use an example from Spafford [1992], that will be useful in the comparison later. If we desired a probability of false positives equal to 0.5\% (an overall error of 0.25\%), and we have a dictionary of 250,000 words, with \( k=6 \) hash functions, we need \( m = 2800 \) Kbits for the hash table, amounting to 350 Kbytes. For words of 8 characters, which we use in our experiments, the dictionary requires 2.25 Mbytes. This is like a compression from 100 to 16, which we indicate with a 0.16 “compression coefficient.”

The error rate in this case is much lower than the one reported in the experiments for BApasswd [Davies and Ganesan 1993], and, in general, can be made as low as desired by using larger hash tables. Thus the technique based on Blum filters is preferable if security requirements are high, and we wish to devote a substantial amount of disk space to the password-checking problem. In fact, for low errors and large dictionaries (e.g., 100 Mbytes), the size of the hash table is considerable. The experiments described later suggest that the technique proposed in the present paper achieves in most cases higher compression and a lower error rate. We also try to explain these findings from a theoretical perspective, based on recent results in computational learning theory [Beimel 1996; Bergadano et al. 1996; Dietterich et al.1996]. The basic problem with Blum filters is that the hash table compression of the dictionary overfits the given data and is bound to a limited predictive power. In fact, Blum filters could hardly be used when training on one dictionary and then testing on different but related dictionaries, as was done in the BApasswd study [Davies and Ganesan 1993], and in one of our experiments with ProCheck.

3. PASSWORD CLASSIFICATION WITH DECISION TREES

We view the training phase of a password checker as a particular machine-learning problem. More precisely, we like to use the dictionaries as sources for positive examples, and learn more concise descriptions that classify passwords as either positive or negative. We also choose to create an explicit set of negative examples by generating random passwords that do not belong to the dictionaries. Examples are given all at once, and no new dictionaries can be added without repeating the training phase. This fits into the standard framework of one-step learning from positive and negative examples. An immense body of literature exists on this topic (for recent
work, see Saitta [1996]). Inductive approaches have also been used in computer security for anomaly-detection applications [Monrose and Rubin 1997].

Moreover, for the learned dictionary descriptions, we chose a decision tree representation. This is done for three reasons: (1) word membership is a simple problem and does not need more expressive formalisms; (2) excellent systems were developed and implementations are available; and (3) decision trees are likely to achieve greater compression on word membership problems because prefixes common to many words need be stored only once. There are many approaches to learning decision trees, and much relevant literature. Here we only review what is needed for our password-checking applications, and the basics that make the paper self-contained.

For our problem, a decision tree may be a procedure that classifies words as either belonging to the dictionary (positive) or not belonging to the dictionary (negative). Words are described by means of so-called attributes, i.e., functions that take a finite number of values. For example, an attribute could have as its value the second letter of a given word, and another attribute could have the length of the word as its value. A node of a decision tree corresponds to some attribute, and the arcs from this node to its children corresponds to particular values of the attribute. Leaves are associated to a classification, positive or negative. A decision tree can then be used for classifying a word \( w \) as follows: we start from the root of the tree and evaluate the corresponding attribute for \( w \), obtaining value \( v \); then we follow the arc labeled by \( v \) and reach the corresponding node; we repeat the procedure until a leaf is reached; and output the associated classification.

As an example, suppose we have a dictionary containing just two bad passwords: “ab” and “cow”. Suppose also that we generate two random passwords: “w31” and “exw”. We label “ab” and “cow” as negative examples, and “w31” and “exw” as positive examples. Obviously, this is just for illustration. In practice, we use dictionaries with million of words, each with 6 characters or more. We now describe these examples with 3 attributes:

- \( a_1 \) equals 0 if the first character is a vowel, 1 otherwise;
- \( a_2 \) equals 0 if the second character is a vowel, 1 otherwise;
- \( a_3 \) equals the length of the word.

Figure 1 shows two decision trees that are both acceptable solutions of this simple classification problem. The tree on the right, for example, first considers the length of the word and then examines the second character to see whether it is a vowel. If the length is 3, and the second letter is not a vowel, we obtain the positive examples “exw” and “w31”, and no negative examples. This is why the corresponding node of the tree is a leaf and is labeled as positive.

A reference system for learning decision trees is the very well known ID3 [Quinlan 1986]. Its basic top-down learning strategy is followed by most
other methods. Initially, the tree is empty and an attribute needs to be selected for the root node. All positive and negative examples are associated to the root node of the tree. Among the possible attributes, ID3 chooses one that maximizes an information-theoretic quantity called the gain.

The gain of an attribute $a$ is computed as follows. Suppose the father node is associated to $p$ positive and $n$ negative examples, with $p + n = t$. Define the information represented by this partitioning of positive and negative examples as $I(p, n) = -(p / t) \log_2(p / t) - (n / t) \log_2(n / t)$.

The information after attribute $a$ has been selected then computed as the weighted sum of the information corresponding to the children nodes: $I(a) = \sum_{i=1}^{s} \frac{t_i}{t} I(p_i, n_i)$, where there are $s$ possible values of $a$, and $p_i(n_i)$ out of the $p$ positive examples have the $i$th value for attribute $a$. Again, $t_i = p_i + n_i$. The gain of attribute $a$ is then defined as $I(p, n) - I(a)$.

After an attribute has been selected, arcs and children nodes are generated for each of the possible attribute values. Sets of examples are associated to these new nodes — more precisely, examples that were associated to the father node are partitioned according to their values for the selected attribute. If a node is only associated to positive (negative) examples, it becomes a leaf of the tree with a corresponding positive (negative) classification label. Otherwise, the whole procedure is repeated on generated nodes, until only leaves remain. Intuitively, the gain selects the attribute that will generate children nodes that are as “unbalanced” as possible with respect to their corresponding classifications, i.e., where the number of associated positive and negative examples is unequal. Since the tree-generation procedure stops when only positive or only negative examples remain, it is clear that the gain criterion will drive the procedure towards that stopping point faster, and will in general produce smaller trees.

An important topic in decision-tree learning that is very relevant to the present study, goes under the name of pruning. Decision trees that are...
learned with the above technique correctly classify all given examples, if the attributes are sufficient to discriminate one class from the other. In other words, the observed error rate is 0. This is the case in Figure 1. However, if the tree is large, it may happen that some leaves, or even large subtrees, are only useful for a limited set of examples. One could eliminate such subtrees, thus increasing the observed error rate and decreasing the size of the tree. Where subtrees are eliminated, nodes that were previously internal become leaves, and the associated classification is computed via a majority rule, i.e., if most examples associated to that node are positive (negative), the node is labeled as a positive (negative) leaf.

In general, tree pruning has always been considered attractive. One reason is, of course, related to the increased simplicity of the classifier. But the most important reason is related to predictive power. It was noticed early on that pruned trees, although they had higher observed error rates, were almost always better when used on new examples drawn from the same distribution. In other words, their observed error rate could be higher, but their expected error rate on future cases is likely to be lower. The reason for these findings is related to overfitting phenomena that are well known in pattern recognition. The theoretical reasons for the same findings are now also well understood, and are related to the expressiveness of the hypothesis space [Vapnik 1982].

A survey of decision-tree pruning techniques is found in Quinlan [1987]. Although we are immediately concerned with compression for the password-checking problem, prediction is not worthless, as we discuss after the experiments. In fact, we would like to reject bad passwords in general, and not just passwords belonging to a specific dictionary that could be incomplete. For both reasons then, that is, compression and prediction, tree pruning has been adopted in our approach.

Another justification for pruning, extremely relevant for the password-checking problem, is related to Rissanen's Minimum Description Length principle [Rissanen 1986], and has been investigated in connection with decision trees by Quinlan and Rivest [1989]. Here we adapt it to the specific password-classification problem. Consider a combined dictionary of good and bad passwords, stored in a two-column file: the first column contains words, and the second column contains their classification, 0 or 1, meaning that they are either good or bad passwords, respectively. Now consider the problem of storing the dictionary in minimum space. One way of doing this is to generate a decision tree for classifying the words, and only then store the first column of the dictionary, plus the decision tree itself. If the dictionary is large compared to the tree, this could save considerable space. However, if the observed error rate is greater than 0, we also have to store a list of exceptions, e.g., as a separate two-column dictionary with the correct classification of words misclassified by the decision tree. Again, if such exceptions are not too numerous, the overall representation of the dictionary could still lead to some compression.

It goes without saying that the above principles fit our application needs. We want to replace a dictionary of bad passwords with a more concise
representation. We then learn a decision tree with the above techniques. If the decision tree has an observed error rate that is not 0, and we list the exceptions in a separate file. This file is searched first when checking user passwords. BApasswd proposes a similar technique for dealing with classification errors [Davies and Ganesan 1993]. We would like the overall space needed for the decision tree and the file of exceptions to be as small as possible.

Although the minimum description length approach is attractive for our application, in our experiments we use the C4.5 system [Quinlan 1993]. In fact, this system has become a standard reference in decision tree learning. It has sophisticated pruning mechanisms and performs well on most applications. Moreover, it is readily available and the implementation is robust. Note that our password-checking application leads to sets of millions of examples, and few learning systems are engineered to work with such data sizes. Actually, even C4.5 gave us some problems of this kind in Experiment 1, below; and we had to partition the set of examples and run multiple training phases, with lower performance results than could be achieved in one-step. In Experiment 2, also described below, the dictionary size could be handled directly by C4.5 without partitioning the training set because most dictionary words were only used for testing.

4. EXPERIMENT 1

We had a large crack dictionary called very big dict (crack version 4.1), with more than 1.5 million words of different lengths, which uses approximately 15 Mbytes of disk space. For implementation reasons, C4.5 could not run with such a large number of positive examples on a Sun Sparc 20 with 80 Mbytes of memory and a large swap area. If we were to apply the method, we would have to split very big dict into about 10 partitions, and use the combined decision trees for password classification. This is unfortunate, as greater compression could very probably be achieved by running the system in just one step, with all the examples. However, the results are already quite good, with about 150,000 words in each dictionary. This could be seen as a practical limitation on the use of decision trees for this purpose, but it is not very strong (see the description of Experiment 2 in Section 5).

Here we report on experiments with three different subsets of very big dict:

(1) bad dict1 contains 145,026 words obtained by truncating characters to 8 words drawn randomly from the larger very big dict;

(2) bad dict2 contains 145,026 words drawn randomly from very big dict that are of length 8;

(3) bad dict3 is the same as bad dict2, but with a different random choice.

Moreover, separate dictionaries of random “good” passwords were generated:
— good dict1 contains 145,026 random 8-character words that do not belong to bad dict1;

— good dict2 contains 145,026 random 8-character words that do not belong to very big dict;

— good dict3 is the same as good dict2, but with a different random choice.

Some comments are in order. First, all words in all dictionaries are 8 characters. This is good for Unix systems; but, of course, the method could be used for passwords of any length. Second, since bad dict1 was obtained with truncations, it has a large number of repeated words. Moreover, the bad passwords are “easy,” in the sense that they are simple variations of truncated words common in English. For instance, bad dict1 includes the consecutive words “directam,” “directeu,” “directin,” “directio,” “directit.” In very big dict passwords longer than 8 characters are much more frequent than passwords of exactly that length. Moreover, passwords in bad dict1 are taken by a fraction of the whole dictionary containing words with at most one special character. Dictionaries bad dict2 and bad dict3, by contrast, do not contain repetitions, and a lot of their words are difficult in the sense that they may be bad passwords if very strict requirements are used, but are certainly not trivial. Consecutive examples are 

\steray 1, 00000000, j-cross-, {ttleden.

The dictionaries good dict1, good dict2, and good dict3 contain random words that certainly look like good passwords (e.g., “tioq!z-”, “> xsa]qf&”, “qm*obnsj”). We view dictionaries 2 and 3 (bad and good) as quite representative of 8-character password-checking problems with difficult dictionaries.

We experimented with a number of different attributes. Attributes are important because they determine which password features are used for classification. All of our attributes, except one — attribute d, explained below — depend directly on the value of some letter in the password. In particular, each experiment uses attributes selected from among the following:

— a1, a2, ... , a8: letters 1 through 8 in the password, described with a value from the set

', ', #, $, %, ', bracket, *, comma, - , dot, / ,
0,1,2,3,4,5,6,7,8,9,
colon, semicolon, < , > , = , ?, @ , \
 a, b, c, d, e, f, g, h, i, j, k, l, m, n,
o, p, q, r, s, t, u, v, w, y, z,
|, other

— b1, b2, ... , b8: letters 1 through 8 in the password, with a value from the set a (vowel), b (consonants n, m, r, l), c (consonants c, g, k, x, j, q, h),
d (consonants t, d, b, p), e (consonants f, v, w, s, z), f (digits), g (other characters);

—c1, c2, . . . , c8: letters 1 through 8 in the password, with a value from the set a (vowel), b (consonants n, m, r, l, c, g, k, x, j, q, h), c (consonants t, d, b, p, f, v, w, s, z), d (digits), e (other characters);

—d: this attribute returns 1 if the word contains at least one character that is not alphanumeric, and 0 otherwise.

For example, for the password “ax00000”, attributes a1 through a8 have values a, bracket, x, 0, 0, 0, 0, 0; attributes b1 through b8 have values a, g, c, f, f, f, f; attributes c1 through c8 have values a, e, b, d, d, d, d, d; and attribute d is 1. In each experiment we used only a subset of the above attributes, and the decision trees we obtained associate nodes to attribute names, and arcs to attribute values. We grouped consonants together as in b1, b2, . . . , b8, considering similarities in letters as phonetic; in trying to reduce the number of groups, we simply merged some of them, obtaining the representations given in c1, c2, . . . , c8. Of course, different attribute choices, or different consonant groupings, were possible, but our choices were sufficient to obtain good results and to compare the different levels of generality in the attribute values. We did not explore other attribute-value representations because we were quite satisfied with our results.

The size of each of the three dictionaries bad dict1, bad dict2, and bad dict3 is 1,305,234 bytes, equal to 145,026 words times 9 (8 characters per word, plus a newline that is needed for words of different lengths anyway). Table I describes each experiment with

—the experiment number;
—the name of the dictionary of bad passwords;
—the name of the dictionary of good passwords;
—the attributes used;
—the size in bytes of the decision tree obtained after pruning;
—the number of positive and negative examples that are not classified correctly; and
—the compression coefficient, i.e., the ratio between the size of the dictionary of bad passwords and the size of the decision tree.

Clearly, for each dictionary, we can choose the best result, i.e., use the attributes that lead to the highest compression and lowest error. The best experiment for bad dict1 is number 1a; the best for bad dict2 is 2f; and for bad dict3 we chose experiment 3f. Note that using a larger character set is good for the first dictionary, while the second and third dictionaries require attributes c1–c8, with a limited set of possible values. We would now like to analyze these experiments further, in an order corresponding to the best attribute choices. In particular, we evaluate another kind of compression
coefficient; the data will be useful in the comparisons of the next section. For each of these three experiments, Table II shows

— the experiment number;
— the size of the dictionary of bad passwords (dsize);
— the size of the tree (size1);
— the first compression coefficient (c1), computed as size1/dsize;
— the error rate (error1);
— the size of a dictionary of bad words that are classified as good (size2);
— the second compression coefficient (c2), computed as (size1 + size2)/dsize;
and
— the error corresponding to c2 (error2), which is one sided and equal to half of error1.

For the last item, we make the assumption that such errors are equally balanced, i.e., half of the errors are false positives and half are false negatives. Then size2 is computed as the number of classification errors divided by two and multiplied by 9 (8 characters plus a newline).

Table II is useful if one wants to evaluate dictionary compression with an error that is just one-sided, such that no words in the dictionary are classified as good. This may be a security requirement that is important in some cases. We immediately see that, with the requirement of one-sided error, the compression coefficient does not change dramatically. In general, the above results show that the method can achieve high compression (from 100 to 3, on the average) with limited error (of the order of 1 per cent).

<table>
<thead>
<tr>
<th></th>
<th>bad words</th>
<th>good words</th>
<th>attributes</th>
<th>tree size</th>
<th>error rate</th>
<th>compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>bad dict1</td>
<td>good dict1</td>
<td>a1-a8</td>
<td>32121</td>
<td>1455 (0.5%)</td>
<td>0.02</td>
</tr>
<tr>
<td>1b</td>
<td>bad dict1</td>
<td>good dict1</td>
<td>a1-a8,d</td>
<td>56561</td>
<td>671 (0.2%)</td>
<td>0.04</td>
</tr>
<tr>
<td>1c</td>
<td>bad dict1</td>
<td>good dict1</td>
<td>b1-b8</td>
<td>49413</td>
<td>1850 (0.6%)</td>
<td>0.04</td>
</tr>
<tr>
<td>1d</td>
<td>bad dict1</td>
<td>good dict1</td>
<td>b1-b8,d</td>
<td>39521</td>
<td>1585 (0.5%)</td>
<td>0.03</td>
</tr>
<tr>
<td>1e</td>
<td>bad dict1</td>
<td>good dict1</td>
<td>c1-c8</td>
<td>43493</td>
<td>1911 (0.7%)</td>
<td>0.03</td>
</tr>
<tr>
<td>1f</td>
<td>bad dict1</td>
<td>good dict1</td>
<td>c1-c8,d</td>
<td>38649</td>
<td>1856 (0.6%)</td>
<td>0.03</td>
</tr>
<tr>
<td>2a</td>
<td>bad dict2</td>
<td>good dict2</td>
<td>a1-a8</td>
<td>122001</td>
<td>6110 (2.1%)</td>
<td>0.09</td>
</tr>
<tr>
<td>2b</td>
<td>bad dict2</td>
<td>good dict2</td>
<td>a1-a8,d</td>
<td>101501</td>
<td>4203 (1.4%)</td>
<td>0.08</td>
</tr>
<tr>
<td>2c</td>
<td>bad dict2</td>
<td>good dict2</td>
<td>b1-b8</td>
<td>60213</td>
<td>4640 (1.6%)</td>
<td>0.05</td>
</tr>
<tr>
<td>2d</td>
<td>bad dict2</td>
<td>good dict2</td>
<td>b1-b8,d</td>
<td>43985</td>
<td>4234 (1.5%)</td>
<td>0.03</td>
</tr>
<tr>
<td>2e</td>
<td>bad dict2</td>
<td>good dict2</td>
<td>c1-c8</td>
<td>49109</td>
<td>3834 (1.3%)</td>
<td>0.04</td>
</tr>
<tr>
<td>2f</td>
<td>bad dict2</td>
<td>good dict2</td>
<td>c1-c8,d</td>
<td>43329</td>
<td>3867 (1.3%)</td>
<td>0.03</td>
</tr>
<tr>
<td>3a</td>
<td>bad dict3</td>
<td>good dict3</td>
<td>a1-a8</td>
<td>122001</td>
<td>6087 (2.1%)</td>
<td>0.09</td>
</tr>
<tr>
<td>3b</td>
<td>bad dict3</td>
<td>good dict3</td>
<td>a1-a8,d</td>
<td>101501</td>
<td>4180 (1.4%)</td>
<td>0.08</td>
</tr>
<tr>
<td>3c</td>
<td>bad dict3</td>
<td>good dict3</td>
<td>b1-b8</td>
<td>60213</td>
<td>4621 (1.6%)</td>
<td>0.05</td>
</tr>
<tr>
<td>3d</td>
<td>bad dict3</td>
<td>good dict3</td>
<td>b1-b8,d</td>
<td>42113</td>
<td>4297 (1.5%)</td>
<td>0.03</td>
</tr>
<tr>
<td>3e</td>
<td>bad dict3</td>
<td>good dict3</td>
<td>c1-c8</td>
<td>49109</td>
<td>3817 (1.3%)</td>
<td>0.04</td>
</tr>
<tr>
<td>3f</td>
<td>bad dict3</td>
<td>good dict3</td>
<td>c1-c8,d</td>
<td>43329</td>
<td>3850 (1.3%)</td>
<td>0.03</td>
</tr>
</tbody>
</table>
We now compare the above results to the BApasswd system [Davies and Ganesan 1993], which uses Markov models, and to the study in Spafford [1992] based on Blum filters. The trigraph occurrence test [Nagle 1988], described earlier, is simple, but has inferior performance. For our decision tree approach, we use the dictionaries \textit{bad dict1}, \textit{bad dict2}, and \textit{bad dict3}, with the results of experiments 1a, 2f, and 3f, which are compared to the experimental results reported for BApasswd on other dictionaries and to the theoretical error estimates for Blum filters.

We expected decision trees to produce good compression because prefixes common to many words are not repeated and all decision-tree learning methods are designed to keep the tree size small. However, high compression is useful only if the error rate is low — otherwise the checker frequently gives wrong advice, or a long list of exceptions is needed. Having to list too many exceptions to the classifier will obviously have a negative effect on compression.

The BApasswd paper reports experiments on several dictionaries. One of these dictionaries (BP5) was used for training and then all were used for testing, i.e., for evaluating the error rate. This aspect of their experiment is relevant for prediction, and is discussed later. For compression, we consider dictionary BP5 only, where BApasswd was trained. BP5 contained 86536 “bad” passwords, and the Markov model obtained rejects all but 1454; that is 1.68\% (observed rate of false negatives). BApasswd does not use negative examples for training, but does produce false positives. The error rate for false positives is estimated in Davies and Ganesan [1993] on a file (GP1) of “good” passwords generated randomly with a set of 95 characters, and is equal to 3.74\%. This is an estimate of the probability of false positives, while for the present experiment with ProCheck, we only evaluated the observed error rate on the given negative examples. The average error, for an equal number of good and bad passwords, is \((1.68\%+3.74\%)/2 = 2.71\%\).

We now evaluate compression. The learned second-order Markov model has a constant size of 175 Kbytes. This is the main problem. The fact that the classifier size is constant means that for small dictionaries it is a waste of space, and for large dictionaries (the usual case) it is insufficient and causes large error rates. Assuming 8 character words, the size of BP5 is 86536*9 = 779 Kbytes. If we keep a two-sided error of 2.71\%, we have a compression coefficient \(c1 = 175/779 = 0.22\). If we want to eliminate false negatives by listing them in an exception file, we obtain an overall size of

<table>
<thead>
<tr>
<th>#</th>
<th>dsize</th>
<th>size1</th>
<th>c1</th>
<th>error1</th>
<th>size2</th>
<th>c2</th>
<th>error2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>1305234</td>
<td>32121</td>
<td>0.02</td>
<td>1455</td>
<td>6548</td>
<td>0.03</td>
<td>728</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(0.5%)</td>
<td></td>
<td></td>
<td>(0.25%)</td>
</tr>
<tr>
<td>2f</td>
<td>1305234</td>
<td>43329</td>
<td>0.03</td>
<td>3867</td>
<td>17406</td>
<td>0.05</td>
<td>1934</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(1.3%)</td>
<td></td>
<td></td>
<td>(0.67%)</td>
</tr>
<tr>
<td>3f</td>
<td>1305234</td>
<td>43329</td>
<td>0.03</td>
<td>3850</td>
<td>17325</td>
<td>0.05</td>
<td>1925</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(1.3%)</td>
<td></td>
<td></td>
<td>(0.66%)</td>
</tr>
</tbody>
</table>
175K + 1454*9 = 188K, yielding a compression $c_2 = \frac{188}{779} = 0.24$, with a corresponding one-sided error of $\frac{3.74\%}{2} = 1.87\%$.

The above data are summarized in Table III, where our results for experiments 1a, 2f, and 3f are also repeated. In general, the compression achieved by BApasswd seems to be poor for dictionaries of this size. Since the Markov model is of constant size, compression improves linearly as the dictionary grows. However, errors also increase, and are already high in the reported experiments. Then, with larger dictionaries, compression coefficient $c_1$ improves, but error and compression factor $c_2$ become worse. If average word length is greater than 8 in BP5, then the data is also better for $c_1$, but worse for $c_2$, depending on the error rate. BApasswd has, however, other advantages, mainly related to prediction and noise handling.

We now evaluate the use of Blum filters, as in Spafford [1992], where much lower error rates can be achieved; but compression is not very good. As mentioned in an example earlier, for a total one-sided error of 0.25% and a dictionary of 250,000 words, we need a hash table of 350 Kbytes, with a compression coefficient ($c_2$) equal to 0.16. This is summarized in line 2 of Table III. If we want greater compression, we need to allow for higher rate of error. For example, for 1% of false positives and an overall one-sided error of 0.5%, we need a hash table of 300K with a compression coefficient of 0.13. This is found in line 3 of Table III. Error rates for false negatives are observed for all the given dictionaries. For false positives, the data for BApasswd is an error estimated on an independent file of random “good” passwords; for Blum filters, it is the theoretical expected probability of false positives. For decision trees, we use the rate of false positives observed on the training set for Experiment 1. Later, in Experiment 2, we did an independent evaluation on a separate file of random words.

Compressions $c_2$ and $c_1$ are the same for the system based on Blum filters because its error is one-sided, i.e., it always rejects words that belong to the dictionary. Then there is no need for a list of exceptions (size2 is 0). For comparable error rates, decision trees achieve considerably higher compression. However, note that, for Blum filters, the rate of false positives is the expected error (on all possible good passwords), while we measured this quantity on the given negative examples. For a fairer comparison, evaluate this error on a separate test set of good passwords, not used during training, as in Davies and Ganesan [1993]. This was done in our Experiment 2, described next. In general, decision trees learned with...
standard tools such as C4.5 tend to have small degradation of performances when moving to previously unseen examples. Blum filters could also be useful in our approach to store the exceptions; that is, instead of listing in a file the words that are misclassified by the decision tree, we could encode them with a Blum filter, and thus reduce the space needed for them. This would improve our c2 compression coefficient. Combined techniques of this kind should be investigated.

5. EXPERIMENT 2

After seeing the results of Experiment 1, we used an independent experimental setting. We obtained significantly higher compression and lower error rates, due to a modified methodology and different attribute choices. The new experiment is described by the items below.

• A dictionary Badpassworddict was considered, taking 28,496,413 bytes and containing 3,215,846 words. The dictionary is oriented to English language users. Word length is between 6 and 12, and words are used without any form of truncation, as was necessary in Experiment 1. More precisely, the dictionary contains:
  - 268409 words of 6 characters;
  - 380912 words of 7 characters;
  - 2364174 words of 8 characters;
  - 59793 words of 9 characters;
  - 55427 words of 10 characters;
  - 47642 words of 11 characters;
  - 39489 words of 12 characters.

• An equally large dictionary Goodpassworddict of random words was generated, with exactly the same word length distribution.

• Both dictionaries are divided into 25 partitions (bad₁, . . . , bad₂₅, good₁, . . . , good₂₅), each with exactly the same word length distribution. Thus, each partition takes 1,139,807 bytes and contains 128,634 words. Although this was not intentional in the design of the random word generation procedure, we observed that, for i between 1 and 25, the intersection of goodᵢ and badᵢ is empty.

• The words in bad₁ were used as negative examples, and the words in good₁ were used as positive examples. A decision tree classifier DTC was obtained by using the following attributes:
  - aₙ (for n between 1 and 12) = value of letter number n in the word, where values are as follows: 1 for vowels; 2 for n, m, r, l, c, g, k, x, j, q, and h; 3 for t, d, b, p, f, v, w, s, and z; 4 for digits; 5 for all other characters;
  - a₁₃ = 0 if the word contains a special character; 1 otherwise;
  - a₁₄ = number of capital letters in the word.
• The words in $\text{bad}_2, \ldots, \text{bad}_{25}$ are used as an independent test set of negative examples, and the words in $\text{good}_2, \ldots, \text{good}_{25}$ are used as an independent test set of positive examples. This is important, as no independent test set is used in Experiment 1.

The results are as follows:

— a decision tree classifier DTC of size 24,161 bytes was obtained;

— a two-sided error of 0.5% was observed on the learning set ($\text{good}_1$ and $\text{bad}_1$);

— a two-sided error of 0.74% was evaluated on the independent test set.

This is far superior to Experiment 1 because it corresponds to a compression of 1000 to 1, with a lower error rate. This improvement is mainly due to a different attribute choice and to an experimental methodology that enabled us to use such a large dictionary (28 Mbytes). Moreover, it should be noted that the error is measured on an independent test set, and may be taken as an estimate of what happens to new passwords that are selected by users.

If a one-sided error is required, one could list the errors for the whole Badpassworddict in an exception file, i.e., the exception file contains all words in Badpassworddict that are classified as good passwords by the decision tree classifier DTC. When deciding whether to accept a user password, ProCheck first classifies it with DTC and rejects it if it is classified as bad. Otherwise, the password is checked for membership in the exception file and rejected if present. This reduces the error to one-sided and is equal to 0.32%, but requires an exception file of 171 Kbytes, increasing compression to 1000 to 7. This is still superior to the results of Experiment 1, and hence to those that may be achieved with other methods. This figure can be improved by compression of the exception file: we were able to reduce it to 104 Kbytes with Unix compress.

General considerations concerning prediction follow from these results. Proactive password checking has the goal of disallowing weak password choices. We would like to find a rule to distinguish the good passwords from the bad. This is a general goal that cannot be tied to a particular dictionary. In other words, it is desirable to have a graceful degradation of performance when moving from the dictionary used for training to other similar dictionaries.

In general, decision tree learners exhibit excellent prediction characteristics [Quinlan and Rivest 1989; Dietterich et al. 1996]. These systems were designed with the explicit goal of prediction, not just for compression. A considerable amount of experimental evidence is found in the literature (recent work can be found in Saitta [1996]), showing that C4.5, or its modifications, only slightly increase their error rate when moving to an independent test set of examples.

The issue, as discussed above, is ill-defined. We have not yet clarified what is meant by "similar" dictionaries. To do this, let us define a
password-checking scenario that is different from the usual one. Suppose there is a huge dictionary $D$ with “all” bad passwords, but suppose also that this dictionary is so large that it could not possibly be stored on disk. Suppose also that a random subset $D_1$ of $D$ is available and can fit in our storage. We could then train our password checker on $D_1$ and evaluate its error $e_1$, also observed on $D_1$. Then, from $D$, we could draw a different set of examples $D_2$, and evaluate the error $e_2$ of the checker on $D_2$. The question we address here is “how close will $e_1$ and $e_2$ be?” The literature cited above contains experimental studies showing that $e_1$ (observed error on training set) and $e_2$ (error on test set) are usually close. The BApasswd study shows that this is also true, to some extent, for the second-order Markov model. This is also demonstrated in this Experiment 2 for decision trees, while our Experiment 1 did not address the prediction issue. For Blum filters the situation is different. All the words of $D_2$ that do not belong to $D_1$ are (wrongly) classified as negative, and error $e_2$ is very high, even if $e_1$ is low. As a classifier, the Blum filter technique “overfits” the positive examples, trying to reject these words at all costs, and losing predictive power.

The problem can be refined further from a more theoretical point of view. Define $e$ as the error of the classifier learned on $D_1$ for the whole dictionary $D$. Suppose also that $D$ is generated by means of a classifier of the same kind (e.g., a decision tree or classifier $C$ with error $e=0$ exists). Then we would like, with arbitrarily high probability, to obtain a classifier $C_1$ that is a good approximation of $C$ with an arbitrarily small error $e$. If we have a procedure that is guaranteed to achieve this goal, we say that this class of dictionary generators is Probably Approximately Correctly (PAC) learnable. It was recently shown in Bshouty [1993]; Bergadano et al. [1996]; Beimel et al. [1996] that decision trees are polynomially PAC-learnable under some general conditions, and the same is true for some variations of probabilistic automata [Beimel et al. [1996]. Although C4.5 is not an algorithm with such characteristics, the methods cited above could be implemented under formal security requirements, with the need for arbitrarily low error, even on previously unseen words. Such methods also allow system administrators to set an error probability threshold for the password checker, and the training phase would then produce an adequate classifier. By contrast, C4.5 runs as a black box, and even the observed error rate cannot be controlled; i.e., C4.5 is trained with an initial set of examples, and a system administrator cannot set an error probability threshold in advance; errors depend on the task and the training set only.

5.1 ProCheck and Bad Noisy Passwords
We also tested our proactive password checker against bad “noisy” passwords, defined as passwords obtained by adding or substituting one noisy character in a bad password [Davies and Ganesan 1993]. This is important in practice because many hackers check for this noise when trying to guess
a password, and many users think adding a noisy character as a special character makes a bad password good.

We have generated some files of “bad noisy passwords,” inserting and substituting a random character, including special characters, in a random location in each word in some partition of our dictionary of bad passwords. We got the results reported in Table IV, where

- **bad*.1ins** is a file of words with a random character inserted in a random location;
- **bad*.1sub** is a file of words with a random character substituted in a random location;
- **bad** is a partition of the big dictionary of bad passwords

Table IV shows that the higher error rate (about 8%) is caused by inserting noise.

Our implementation of the proactive password checker, ProCheck, checks some attributes of the password candidate chosen by the user before submitting it to the classifier. Particularly important is the check that takes the proposed password and generates a dual one where all the uppercase characters are transformed into lowercase, and vice versa (the special characters remain unchanged). Then both words so obtained are submitted to the classifier and ProCheck considers the original candidate to be a good password only if both of them are accepted. Performing this check we get an even better result, as reported in Table V, in fact the higher error rate is about 6%, better than the error rate observed by BApasswd (8%), and is the only approach we know of able to deal with bad noisy passwords.

It is worth noting that the improvement is paid for by a bigger error, in which good passwords are mistakenly considered bad. The extent of this error is about the same as the improvement in the other direction. In many environments, however, it is acceptable to slightly increase the probability

<table>
<thead>
<tr>
<th>file of bad words</th>
<th>good</th>
<th>bad</th>
<th>error rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bad_1284</td>
<td>1284</td>
<td>127349</td>
<td>0.99818089</td>
</tr>
<tr>
<td>bad_1284.1sub</td>
<td>5886</td>
<td>122747</td>
<td>4.5757731</td>
</tr>
<tr>
<td>bad_1284.1ins</td>
<td>10405</td>
<td>118228</td>
<td>8.0889041</td>
</tr>
<tr>
<td>bad_1279</td>
<td>1279</td>
<td>122735</td>
<td>0.99429389</td>
</tr>
<tr>
<td>bad_1279.1sub</td>
<td>5972</td>
<td>122662</td>
<td>4.6426295</td>
</tr>
<tr>
<td>bad_1279.1ins</td>
<td>10456</td>
<td>118178</td>
<td>8.1284886</td>
</tr>
<tr>
<td>bad_1246</td>
<td>1246</td>
<td>127388</td>
<td>0.96863971</td>
</tr>
<tr>
<td>bad_1246.1sub</td>
<td>5918</td>
<td>122716</td>
<td>4.6006499</td>
</tr>
<tr>
<td>bad_1246.1ins</td>
<td>10468</td>
<td>118166</td>
<td>8.1378174</td>
</tr>
<tr>
<td>bad_1265</td>
<td>1265</td>
<td>127369</td>
<td>0.9834103</td>
</tr>
<tr>
<td>bad_1265.1sub</td>
<td>5968</td>
<td>122666</td>
<td>4.6395199</td>
</tr>
<tr>
<td>bad_1265.1ins</td>
<td>10570</td>
<td>118064</td>
<td>8.2171121</td>
</tr>
</tbody>
</table>
of rejecting a good password if this allows decreasing the probability of accepting a weak password.

6. CONCLUSIONS

All of the proposed password-checking methods, including the one presented here, are very efficient at the time the user password is checked. For all of the above methods, the time needed to check the acceptance of a user password does not directly depend on the size of the dictionary. Complexity only depends on the given password.

With our decision trees, one may not even need to inspect the whole password, and checking would be even faster. For passwords that are clearly good or clearly bad, only a few attributes may suffice, and the corresponding path in the decision tree could be shorter than the password length. For instance, a password that begins with "%@3$" would be accepted regardless of the other characters. A Unix password starting with "micro" would be rejected without waiting for the other 3 characters, and the user would be informed immediately through an appropriate interface. Techniques are also available [Martelli and Montanari 1978] for efficiently optimizing decision trees with respect to the average classification cost.

We have evaluated our method and results with respect to prediction, and found them superior to Markov models and Blum filters. However, there may be other advantages to Blum filters. First, their performance does not depend on the kind of dictionary. In contrast, inductive approaches, such as the one presented here and BAPassword, may vary when using different dictionaries. More precisely, dictionaries of "bad" passwords that actually look like random character strings may be more difficult. As a consequence (this is also an advantage of Blum filters), using inductive approaches to work on encrypted dictionaries may be difficult. Thus, dictionary encryption may be considered a useful security measure.

In fact, the dictionary of bad passwords may be used as a list of words that are not worth trying when using brute force methods. If a password checker is available to an attacker and the checker was trained on a

Table V. Bad Noisy Passwords—Experimental Results (2)

<table>
<thead>
<tr>
<th>file of bad words</th>
<th></th>
<th>good</th>
<th>bad</th>
<th>error rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bad21.bad</td>
<td></td>
<td>1119</td>
<td>127515</td>
<td>0.86991674</td>
</tr>
<tr>
<td>bad21.1sub</td>
<td></td>
<td>5402</td>
<td>123232</td>
<td>4.1995118</td>
</tr>
<tr>
<td>bad21.1ins</td>
<td></td>
<td>7664</td>
<td>120969</td>
<td>5.9580357</td>
</tr>
<tr>
<td>bad14.bad</td>
<td></td>
<td>1116</td>
<td>127518</td>
<td>0.86757778</td>
</tr>
<tr>
<td>bad14.1sub</td>
<td></td>
<td>5488</td>
<td>123146</td>
<td>4.2663681</td>
</tr>
<tr>
<td>bad14.1ins</td>
<td></td>
<td>7690</td>
<td>120944</td>
<td>5.9782017</td>
</tr>
<tr>
<td>bad8.bad</td>
<td></td>
<td>1073</td>
<td>127561</td>
<td>0.8341496</td>
</tr>
<tr>
<td>bad8.1sub</td>
<td></td>
<td>5428</td>
<td>123206</td>
<td>4.2197242</td>
</tr>
<tr>
<td>bad8.1ins</td>
<td></td>
<td>7754</td>
<td>120880</td>
<td>6.0279553</td>
</tr>
<tr>
<td>bad3.bad</td>
<td></td>
<td>1059</td>
<td>127575</td>
<td>0.82326601</td>
</tr>
<tr>
<td>bad3.1sub</td>
<td></td>
<td>5477</td>
<td>123157</td>
<td>4.2578168</td>
</tr>
<tr>
<td>bad3.1ins</td>
<td></td>
<td>7880</td>
<td>120754</td>
<td>6.1250076</td>
</tr>
</tbody>
</table>
plaintext dictionary, it could be used for the same purpose. Assuming that the encryption function is not available to the attacker, training the checker on a dictionary of encrypted passwords obviates this problem. User passwords are first encrypted and then evaluated with the checker.

Moreover, systems based on Blum filters can be easily extended when the system administrator wants to add new bad passwords. The new bad passwords need only be given to the hash functions, and the corresponding bits in the hash table will be set. Our system is not incremental, and if new passwords are added, the whole training phase has to be repeated. This is not as easy as it is for hash tables, e.g., Experiment 1 required about 10 minutes of training on a Sun Sparc 20.

Finally, we would like to comment on the performance of the methods with "noisy passwords," i.e., passwords that would belong to the dictionaries after one of their characters is changed, inserted, or deleted. The paper on BApasswd by Davies and Ganesan [1993] deals with this problem well, showing experimentally that the performance of BApasswd does not change dramatically when using noisy words. We have not done similar experiments, but expect similar results for decision trees, since the issue is related to prediction, as discussed previously. By contrast, we expect Blum filters to accept noisy words, for the same reasons related to prediction. In general, if rejecting noisy dictionary words is important, we could check all relevant variants of a user password, and reject the password whenever one of its variants is rejected. This can be done easily, as all cited methods are extremely efficient when checking a password.

The implementation of the proposed method is publicly available at http://maga.di.unito.it/security/resources/ProCheck.html. This software corresponds to that used in Experiment 2:

— a 24 Kbyte decision tree classifier, obtained from a 28 Mbyte dictionary;

—an exception file of 50 Kbytes, obtained after filtering a redundant exception file of 171 Kbytes. Such redundancy is due to the fact that some exceptions (words in the dictionary that are nevertheless accepted by the DTC classifier) are actually rejected by other checks implemented in the complete package.

The distribution is complete with documentation and installation procedures for various platforms. The basic decision tree classifier DTC that we obtained from the 28 Mbyte dictionary is provided "as is," so it may be used directly on passwords that individual users want to check. For a given user password $w$, DTC($w$) will output either an "accept" or a "reject" message. The decision tree DTC is also included in our top procedure, ProCheck($w$), and it does the following:

— rejects $w$ if $|w| < 6$;

— rejects $w$ if it is a variant of login name or other user identification parameters available from the chosen operating system;

— rejects w if DTC rejects w;

— rejects w if DTC rejects w' obtained from w by turning letters from uppercase to lowercase and vice-versa.

Individual users may then run ProCheck to see offline whether their password is well-chosen according to our criteria and dictionary. For system administrators who want their users to run ProCheck when selecting or changing passwords, our distribution includes passwd patches for different Unix platforms (SunOS, OSF1 for DEC Alpha, HP Unix, Linux). The distribution is found at <http://security.unito.it/security/resources/ProCheck.html>. At the time of this writing, a Windows NT version for ProCheck is being tested.

The reported experiments provide evidence that, with respect to previous approaches, and for comparably low error rates, much higher compression is achieved with ProCheck. After training is completed, password checking can be done in a very efficient way, and the method is used with dictionary sizes that are certainly sufficient for practical purposes. ProCheck is also attractive for its efficiency and its predictive power on unseen password dictionaries.

ACKNOWLEDGMENTS
The authors would like to thank Roger Needham for his helpful comments on a draft version of this paper.

References


High Dictionary Compression for Proactive Password Checking


Received: July 1997; revised: November 1997; accepted: February 1998