Hyppocrates: A new Proactive Password Checker

Carlo Blundo[1], Paolo D'Arco[1], Alfredo De Santis[1], and Clemente Galdi[2]

[1] Dipartimento di Informatica ed Applicazioni
Università di Salerno, 84081 Baronissi (SA), Italy
e-mail: {carblu, paodar, ads}@dia.unisa.it

[2] Computer Technology Institute and Department of Computer Engineering and Informatics
University of Patras, 26500 Rio, Greece.
e-mail: clegal@ceid.upatras.gr

January 16, 2003

Abstract

In this paper we propose a new *proactive password checker*, a program which prevents the choice of easy-to-guess passwords. The checker uses a decision tree, constructed applying the Minimum Description Length Principle and a Pessimistic Pruning Technique. Experimental results show a substantial improvement in performance of this checker compared to previous proposals. Moreover, the whole software package we provide has a user-friendly interface, enabling the system administrator to configure an *ad hoc* password proactive checker, in order to satisfy certain policy requirements.

Key words: Tools for Computer Security, Proactive Password Checking.

1 Introduction

The rapid growth on the Internet of software applications with high security requirements has increased the need of authentication tools and identification protocols for checking the access to shared resources. Among these ones, the old password approach is still the more frequently used for its simplicity.

When a user wants to identify himself to a system, during an interactive checking identification protocol, he types on a keyboard a secret word, the *password*, as an answer to an identification query. The system usually stores in a database some information encrypted using the password as a key. When it receives the password, it re-computes the encrypted information and compares this with the stored one. If they match, the access is granted; otherwise it is refused. This mechanism is for example used in Unix-like operating systems [21].

Of course, the scheme is secure if the user keeps secret his own password. However, it is not enough. In the following sections we will try to explain why the secrecy of the password does not guarantee access security looking back at previous works on this subject and describing our solution.

1.1 Password Choice: The Issue of Security

Network security can be attempted in various ways. The violations can be done by physical users as well as by malicious softwares. *Unauthorized accesses* or *illegal uses* of system resources, *virus*, *worms*, and *Trojan horses* are all examples of this multi-aspects issue. A detailed survey of these different topics can be found in [21].

In this paper, we want to focus our attention on *access control*. Here, unauthorized users, called *hackers or crackers*, try to log on a machine. Once logged in, these intruders can behave in several ways. Anderson has done a categorization of the intruders based on the behaviour once logged in. For details, see [1].

The main way used by the hackers to log on a machine is by mean of a user's password. This is possible not only when the user accidentally discloses his password but even when it is "easy" to guess. Indeed, the users frequently choose passwords that are pronounceable and simply to remember. Experimental results, reported in [12], show that many users choose short passwords or trivial ones, and passwords that belongs to well know dictionaries. Thus, an hacker, having a copy of the file containing the encrypted information used by the system during the login process, can try off-line to "guess" one of the passwords in order to match an item of the file. Indeed, if such a match is found, the guessed password enables him to gain access to the system. Since the set of passwords that the users effectively choose is small, it can be exhaustively checked in a reasonable time. Excellent softwares to accomplish this task have been developed in the recent years and are available on-line [13].

Moreover, the problem of choosing good passwords is not restricted to access control of network systems. Passwords are often used to protect *private information*. Simple examples can be the use of a password to protect cryptographic keys, stored on the local disk of the user or password-authenticated key agreement schemes [8, 11].

A more complex example can be the use of a password in systems like Kerberos described in [15]. Even in these cases it is needed to use good passwords.

Therefore, to increase the security level of a password based system, we have to find *a method* to reduce the efficacy of the *exhaustive search attacks*. This goal can be achieved if users are not allowed to choose easy-to-guess passwords. Indeed, if the password can be any element of a given set of big size, then an exhaustive search cannot be afforded in a reasonable time.

We want to stress that it is not always true that longer passwords are stronger than shorter ones with respect to this kind of attacks. It is sufficient to recall that the number of possible passwords composed by exactly *eight* characters randomly chosen among (upper and lower case) letters is $52^8 \simeq 5.3 \times 10^{13}$. Moreover, and unfortunately, the users do *not randomly* choose their passwords but, as stated above, they try to choose pronounceable or easy to remember ones. This further reduces the search space for exhaustive search attacks. On the other hand, if we *force* the user to choose the password using the characters reported in Table 2, the number of possible length *seven* is $94^7 \simeq 6.4 \times 10^{13}$, while the number of length *eight* passwords over the same alphabet is $94^8 \simeq 6.0 \times 10^{15}$.

1.2 Previous Work

The problem of using good passwords has been studied in several papers. [4, 5, 6, 3] has deeply analyzed this problem. He has given a series of hints about password subsets and easily transformations on words of a dictionary, usually applied by the users to generate a password, that should be avoided since they are easy to guess.

So far, four techniques have been proposed to eliminate easy-to-guess passwords:

- *User education.* Users can be told the importance of using hard-to-guess passwords and can be supported with guidelines for selecting good ones. However, this approach has given poor results since the user can avoid to follow these hints and can continue to use easy-to-guess passwords.

- *Computer-generated passwords.* With this approach the user has no choice and the password is assured to be a random element from a big set. The drawback is that such a password could be difficult to remember. Hence, the user could write it on a paper and the consequences can be unpredictable.

- *Reactive password checking.* Periodically checks are done by the system administrator to find guessable passwords. These checks are performed with password cracker programs. All accounts of users which have been cracked are suspended until they change their passwords. Of course, these checks are resources consuming and do not avoid the possibility that between two consecutive checks some users could utilize easy-to-guess passwords. An attack in this period of time can be successful.

- *Proactive password checking.* A proactive password checker is a program that interacts with the user when he tries to change his own password. The proposed password is checked and the change is allowed only if it is hard-to-guess. Moreover, if it is an easy-to-guess password the checker provides some hints to the selection that can be helpful for the user. The philosophy on which these programs are based on, is that the user has the possibility of selecting a password but the system enables the selection only of non trivial ones. This approach is the most promising and is the one that we are going to pursue in the next sections.

To simplify our discussion, we say that a password is a *weak* or *bad* one if it is easy-to-guess; otherwise, we say that it is a *strong* or *good* one.

A proactive password checker conceptually is a simple program. It holds a list of weak passwords that must be rejected. When the user wants to change his password, it checks for membership in the list. If the password is found the substitution is not enabled and a short justification is given; otherwise, the substitution is allowed.

However, a straightforward implementation of such a program is not suitable for two reasons: the list of weak words can be very long and, hence, must be stored in the secondary memory. Then, the time to check membership can be high. Since the check works in an interactive fashion, a long wait of the user cannot be accepted.

Various proactive password checkers that aim to reduce the *time* and *space* requirements of the trivial approach have been proposed.

A first model is described in [14]. In a set up phase, the checker constructs a *table of trigrams*, scanning the words of a "weak" list/dictionary. When invoked for test membership on a given password, it checks its trigrams. If at least two trigrams of the password do not belong in the table, the password is accepted; otherwise, it is refused. With this system all the passwords in the dictionary are rejected.

In [10], a similar system has been proposed. The dictionary of weak words is used to realize *a second order Markov chain.* The behaviour of this system can be simply understood if we visualize the chain with an automata. Indeed, each state corresponds to a digram and each outgoing arc is labelled with a letter. Moreover, at each transition is associated a probability, computed using the frequency of that trigram in the dictionary of weak words. The checker, starting from the initial state, when invoked on a password, realizes a sequence of transitions determined by its trigrams. During this transition stage it computes the overall probability of the password. If this probability is greater then a fixed threshold, the password is rejected. Otherwise, it is accepted. In this case, the membership test is simply a *measure of likehood* with the words described by the Markov chain. This model reduces space and time requirements, since the size of the automata is small and the time for checking a password depends solely on its length. However, it accepts a small number of dictionary words as well as refuses some words which do not belong to the dictionary.

Another interesting model comes from [20], who realized a proactive password checker using the Bloom's filters described in [7]. A Bloom filter of order k is a set of k independent hash functions. Each of them maps a password to an integer between $0, \ldots, N-1$. Given a dictionary of weak passwords, the checker can be constructed setting up a *boolean matrix* with N rows and k columns. Each row corresponds to an integer while each column corresponds to an hash function. All the entries of the matrix at the beginning are zero. Then, they are filled in as follows: for each word in the dictionary, the k hash functions are evaluated and the entry (i, j) is set to 1 if the j-th hash function gives i as output. Once the checker is invoked for test membership, the hash functions are evaluated. If all the corresponding entries are set to 1, the password is rejected, otherwise it is accepted. With this model *all* the passwords of the dictionary are discarded, while a password which does not belong to the dictionary is rejected if all the hash functions output integer values such that the corresponding entries of the boolean matrix are equal to one.

The drawback of this scheme is that the boolean matrix can have a "dramatic" memory requirement and, hence, cannot be stored in the main memory. However, the scheme is an incremental one. If we want to add another password to the filter at a later time, we have simply to evaluate the k hash functions and set the corresponding entries of the matrix.

All these models are an improvement against the straightforward scheme. However, as showed in [2], they have a *low predictive power* when tested on new dictionaries. Indeed, a desirable feature of a proactive password checker is the ability of correctly classifying passwords which do not belong to the initial set. We would like a low *false negative* error rate, i.e., a small number of weak passwords classified as strong ones, and a low *false positive* error rate, i.e., a small number of strong passwords classified as weak ones.

To this aim, an interesting approach to design a proactive password checker seems to be the one applied in [2]. Here, the problem of password classification is viewed as a *Machine Learning Problem.* The system, in a training phase, using dictionaries of examples of weak and strong passwords, *acquires the knowledge* to distinguish weak passwords from strong ones. This knowledge is represented by a *decision tree.* During the checking phase, the tree is used for classification. The experimental results reported in [2] show a meaningful enhancement on the error rates compared to the previous solutions.

Since the approach seems promising, we have followed the same direction. We have tried to increase the power of such a checker exploring another key idea of machine learning in the construction of the decision tree: the *Minimum Description Length Principle (MDLP).* This principle essentially establishes that the best theory to describe a data set is the one that minimizes the sum of the length of the theory and of the length of the data when coded using the theory. Based on the promising results presented in [18], concerning with the inference of decision trees using the MDLP, we have realized a prototype implementation and have done many tests to measure the performance of this new checker. The results obtained confirm the validity of this choice.

A "folklore" note. We have chosen the name HYPPOCRATES for two reasons: first because it recalls the famous doctor of the ancient times[1]. Then, because the name HYPPOCRATES appears into the logo of our University to remember the `Schola Medica Salernitana`, one of the first European "Academic" Institutions, supposed to be among the ancestors of the modern Universities.

Organization of the paper. In Section 2 we introduce the concept of classification with decision tree, briefly overview some pruning techniques which can increase the predictive power of the tree in classification tasks, and give a short description of the proactive password checker realized in [2]. In Section 3 we recall the principles on which we have realized HYPPOCRATES, our proactive password checker. Finally, in Section 4, we report experimental results and comparisons with previous proposals.

2 Classification with Decision Trees

In this section we describe the decision tree technique for classification used in the setting of proactive password checking in [2]. Moreover, we briefly outline some common pruning techniques applied to increase the predictive power of decision trees.

2.1 Decision Trees

A decision tree can be thought of as a mean to make decision. The root of the tree represents the starting point of the decision process while each leaf corresponds to a decision instance. To each internal node of the tree is associated an *attribute* that can be evaluated on an element that is undergoing the classification process, and which can assume many *values*. The values of the attribute are associated with the arcs outcoming from each node. A *decision* on a given element consists in a *path* from the root to a leaf determined by the values assumed by the attributes associated to each node along the path. More precisely, if the attribute associated to the current node can assume k different values, and the element we are considering assumes the i-th one, then the decision process will continue from the i-th child of the current node.

Since the problem we want to solve is *classification of passwords*, our aim is to realize a decision tree which can be used to establish if a password is a weak or a strong one. Hence, each leaf of the tree must correspond to a weak class of words or to a strong one. The example reported in Figure 1 clarifies our discussion.

There are several methods to construct a decision tree. One of the most interesting is to consider the construction as a Machine Learning Problem. With this approach, we construct the tree by means of a *training* process in which a *set of examples* of weak and strong passwords is used to set up the tree. It is necessary to establish the *set of attributes*

[1] Actually, our system does not look after the ills of a bad password choice, it tries to do prevention avoiding bad choices and giving helpful hints to the users for good ones.

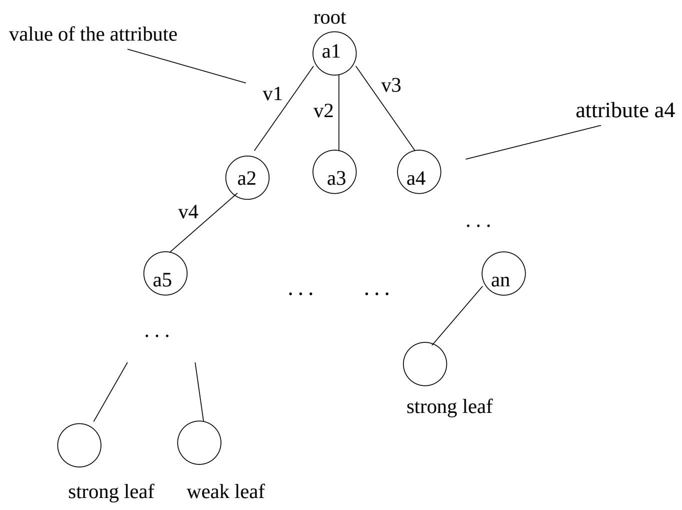

Figure 1: Example of a decision tree for password checking

that must be evaluated in each node of the tree and a *criterion* to associate the attributes to the nodes. These are critical steps in the training phase. Indeed, the choice of examples, the choice of the set of attributes, and the criterion used for attribute selection at each node determine the performance of the tree (i.e., its size and its predictive power) when applied in classification tasks.

The learning process of a decision tree can be described as follows: initially the root contains all the examples of the training set. Then, the training procedure tries to select an attribute for the root to partition the set of examples in subsets, according to an attribute selection strategy that optimizes a certain complexity measure. When this attribute is found, the partition is done and each "new" subset of examples is associate to a root's child. The procedure is applied recursively on the children. The partition process on a node halts when it contains only elements of a given class or when there are no attributes to use for subsequent partitions. This means that, when the training process is complete, some leaves can store elements from different classes. The class of such a leaf is determined as the class of the greatest number of elements of the same type. Usually, in classification tasks which require an *exact* classification of the training set, this problem is bypassed using a file of exceptions in which can be stored all the misclassified elements.

2.2 Pruning Techniques

Once the training phase has been completed, the decision tree can be used by the proactive password checker to establish if a password chosen by the user is weak or strong. If the password we are going to check belongs to the training set, then its classification is surely correct[2]. However, if the password does not belong to the training set, its decision process proceeds along a path defined by the values assumed by the attributes on the password in each node. The hope is that the tree has stored *meaningful features* of the training set so that it is able to correctly classify a new password which shows characteristics similar to the examples seen during the training phase. In other words, our hope is that the decision tree has a good

[2]Assuming that, if the password is misclassified by the decision tree, then it belongs to the file of exceptions.

predictive power as well. However, notice that the decision tree can be *strongly dependent* from the training set. This means that we have a refined classification of the examples but a poor performance on unseen elements. To avoid this behaviour, we can simplify the structure of the tree using a *pruning technique*. The idea is that if we cut a subtree and replace it with a leaf, our tree can show a small error on elements of the training set, (i.e., misclassification) but it can gain predictive power on unseen dictionaries of words.

Many different techniques are known in literature to prune a decision tree. Among these, three are frequently used: *Cost-Complexity Pruning* [9], *Reduced Error Pruning* [16], and *Pessimistic Pruning* [16]. Let us give a quick look at all of them.

Cost-Complexity Pruning. It works in two phases: in the first phase a sequence of trees $T_0, T_1, \ldots, T_k$ is generated. T_0 coincides with the starting tree while each tree T_{i+1} is obtained from T_i by a *substitution* of one or more subtrees with leaves. The process continues until T_k is reduced to a single leaf. More precisely, let T be a decision tree used to classify each of the N objects of the training set, and let E be the number of misclassified objects. If $L(T)$ is the number of leaves of T, we define the measure *cost-complexity* of T as

$$\frac{E}{N} + \alpha L(T)$$

for some value of the parameter α. Suppose that we substitute a subtree S of T with the best leaf, i.e., the *most representative* of the classes contained in S. This new tree will misclassify M more objects of the training set but it has $L(S) - 1$ leaves less than T. Notice that the cost-complexity of this new tree is exactly the same of T if $\alpha = \frac{M}{N(L(S)-1)}$.

In the first phase, the pruning procedure, to generate T_{i+1} from T_i, examines each subtree of T_i that is not a leaf, looking for the best value of α. When this value is found, the subtree is substituted with its best leaf.

In the second phase the cost-complexity measure is not considered. The procedure tries to choose one of the trees $T_0, T_1, \ldots, T_k$ testing their performance on a *test set* different from the training set. If the test set has N' objects and the *minimum observed error* on the tree T_i is E', then the procedure computes the standard error given by

$$stderr(E') = \sqrt{\frac{E'(N' - E')}{N'}}$$

and chooses the tree T_i with *minimum size* such that its error is less than $E' + stderr(E')$.

This pruning technique has some problems: it is not clear where are the benefits of using cost-complexity against other standard measures and, why this measure is used in the first phase but is discarded in the second one. Moreover, the technique requires a test set different from the training set.

Reduced Error Pruning. This technique is simpler than the previous one. It still uses a test set, different from the training set, to evaluate the performance of the tree. It works as follows: let T be the tree generated during the training phase. For each subtree S of T the procedure counts the number of errors on the test set obtained if the subtree is substituted with its best leaf. If this number is less than the one of the starting tree and the considered subtree does not contains a subtree with the same property, then S can be replaced with the leaf.

Notice that even this procedure generates a sequence of trees. However, in this case the final tree is more refined of the starting one on the test set and is the smallest tree with that accuracy.

The problem of this technique consists, again, in the use of a separate test set. Moreover, isolate classes do not present in the test set can be cat. Hence, the tree *can loose* important information.

Pessimistic Pruning. To understand the logic of this technique notice that, during the training phase, an internal node of the tree contains well classified objects and misclassified ones. For example, suppose that in a node with K objects, J are misclassified and the remaining $K - J$ are well classified. The objects of this node are therefore partitioned in a subsequent step of the procedure. In the pruning phase, starting from a decision tree, we have to substitute a subtree with one leaf. We could decide to cut a subtree if the ratio $\frac{J}{K}$ is below a certain threshold. However, this is not a good strategy since this ratio does not give information about the *overall structure* of the tree. Therefore, we need a more complex approach.

Let S be a subtree of T which contains $L(S)$ leaves and let $E = \sum J$ and $N = \sum K$ be, respectively, the number of misclassified objects and the total number of objects of the leaves in the whole tree, respectively. The pessimistic pruning approach supposes that S will produce a bad classification of $E + \frac{L(S)}{2}$ elements in a set of N ones that do not belong to the training set. Denote by E' the number of misclassifications obtained if S is replaced with one of its best leaves. The pessimistic pruning technique replaces S if $E' + \frac{1}{2}$ is less than $(E + \frac{L(S)}{2}) + stderr(E + \frac{L(S)}{2})$.

To clarify the above procedure, suppose that we have two classes, and assume that S is a subtree with the following characteristics: $N = 4000$, $L(S) = 6$, and $E = 0$. Moreover, the objects which belong to the first class are 3998 while there are only two objects in the second class. This means that there is at least a leaf that classifies the elements of the second class and the other leaves that classify the 3998 elements of the first one. Now, the guessed error on N cases, not present in the training set, is $E + \frac{L(S)}{2} = 3.0$ with $stderr(E + \frac{L(S)}{2}) = \sqrt{\frac{3*3997}{4000}} = 1.73$. If S is replaced with a leaf of the first class the error on the training set is equal to two and, hence,

$$E' + 1/2 = 2 + 1/2 = 2.5 < \left(E + \frac{L(S)}{2}\right) + stderr\left(E + \frac{L(S)}{2}\right) = 3.0 + 1.73$$

Therefore, according to the above rule, the pruning of S can be done.

2.3 Proactive Password Checking with a Decision Tree

[2], first suggested the use of decision trees in the design of a proactive password checker. The decision tree stores information about the weakness or the hardness of the passwords used as examples in the training process. It can be viewed as a *high compress representation* of the dictionary of examples, where compression is obtained since prefixes common to many words need to be stored only once.

The criterion chosen by [2] to set up the decision tree was to maximize an information-theoretic measure called *the gain*. The idea is the following. Suppose that we have a set of attributes A. We have to select an element $a \in A$ for each node of the tree. Suppose that the training set is composed of p strong examples and n weak examples, and let $t = p + n$. When the selection procedure starts, all the elements of the training set are associated to the root of the tree. The *information* represented by the partition of the t examples is defined as:[3]

$$I(p, n) = -(p/t)Log(p/t) - (n/t)Log(n/t).$$

Although, if a is the selected attribute, we define

$$I(a) = \sum_{i=1}^{s}(t_i/t)I(p_i, n_i),$$

where s are the possible values that a can assume, and p_i (resp., n_i) out of the p (resp., n) are strong (resp., weak) examples having the i-th value for attribute a, and $t_i = p_i + n_i$. The *gain* of the attribute a is then defined as $I(p, n) - I(a)$. Therefore, for each node, this procedure partitions the examples according to the attribute that maximizes the gain.

Notice that [2], in their checker, use a well know system for learning decision trees, called $C4.5$ [17], which implements the above criteria. This system optimizes the constructed tree, to enhance predictive power on data sets different from the training set, with a pruning operation. This optimization is realized applying the Pessimistic Pruning Technique described in the previous paragraph. Once the decision tree has been constructed, the checker is simply implemented. Indeed, it is a program that, when invoked, does a set of preliminary checks on some basic features of the password (e.g. length,...) and, then, uses the decision tree to decide class-membership.

3 HYPPOCRATES: The Decision Tree

So far we have explained how to use decision tree techniques for password classification, and we have pointed out that it is necessary to choose a criterion for attribute selection to construct a decision tree. Moreover, since the constructed decision tree strictly depends on the training dictionaries, we have also described some pruning techniques "to simplify" its structure in order to enhance the predictive power. In this section we describe the choice we have done in the design of our system. First of all, we recall the minimum description length principle.

3.1 Minimum Description Length Principle

The Minimum Description Length Principle has been introduced by [19]. Conceptually, it says that the best theory to describe a data set is the one that minimizes the sum of two elements:

- the size of the theory
- the length of the data when coded using the theory

In our setting this principle essentially means that the shortest tree (i.e.,the one who minimizes the *description-length*) is the best in order to well classify the training set and to minimize the error on unseen sets of examples.

[3] In this paper all the logarithms are to the base 2.

An intuitive justification of this principle, applied to the decision trees environment, can be achieved if we consider the problem of constructing a decision tree as a *communication problem* in which we want to minimize the communication cost[4]. To this aim, we consider a scenario in which there are two users, A and B. They partially share a table, like the one presented in Table 1, with several columns. In the first column there are the words (weak and strong). The subsequent columns are indexed by some attributes of the words and are filled in with the corresponding values of the attribute for each word. The last column contains the classification of the words and it is stored *only* by A. The problem that A has to solve is *to send* this last column to B with the *minimum number* of bits.

words	attr_1	...	...	attr_n	class
password	val_1	...	...	val_j	W
proactive	val_3	...	...	val_1	W
...	...	...	...	...	...
...	...	...	...	...	...
x1[!T?	...	...	...	...	S
checker	val_s	...	...	val_r	W

Table 1: Table stored by A and partially shared with B

Suppose that this column has the following form:

$$W, W, S, S, S, W, S, W, S, S, S, S, S, W$$

where W corresponds to weak and can be encoded with a bit equal to 0 and S corresponds to strong and can be coded with a bit equal to 1. If we send directly this encoding of the column, then we need to transmit 14 bits. In general, we can apply the following transmission strategy: suppose that the sequence we have to send has length n. If the sequence has k bits equal to 1, we can send the value k and an index that establishes which of the possible $\binom{n}{k}$ k-weight sequences corresponds to the given sequence. The total number of required bits is quantified by the function

$$L(n, k, b) = Log(b + 1) + Log\binom{n}{k}$$

In this formula b is a fixed constant greater than k, big enough to ensure that $Log(b+1)$ bits can represent[5] the value of k.

In the above example, with $b = 15$, since $n = 14$, and $k = 9$ it holds that $L(14, 9, 15) = 14.874$. Hence, in this case this coding technique is not cheaper than the trivial transmission. However, we can do better. Notice that we can have an enhancement in our transmission problem using the *common table*. For example, suppose that with respect to a *given attribute*,

[4]For more details on the material of this section, the reader is referred to [18].

[5]For a binary classification problem b can be fixed to $\frac{n+1}{2}$. Indeed, as we will see later, each string is the representation of a subset of the objects that must be classified belonging to the partition generated by a given attribute. Now, associating to the subset a *default class*, we can see the 1 of the string as the *exceptions* with respect to the default class. Since we can choose the default class, we can always code the exceptions with a string whose weight is less than $\frac{n+1}{2}$. In general, when an upper bound on the number of one of the string cannot be predicted a priori, it is fixed to n.

which takes two values, the classified words can be split into two sets: W, W, S, S, W, S, W, and S, W, S, S, S, S, S. If we apply the previous strategy to the two sequences, then we have that the total transmission cost is equal to

$$L(7,3,3) + L(7,1,3) = 11.937$$

which is an improvement compared to the trivial transmission approach. The key point in this example is that if we partition the starting set according to a given attribute, the overall transmission complexity can be reduced.

The function $L(n,k,b)$ can be approximated applying the Stirling function. More precisely, we have that

$$L(n,k,b) = nH\left(\frac{n}{k}\right) + \frac{4Log(n)}{2} - \frac{Log(k)}{2} - \frac{Log(n-k)}{2} - \frac{Log(2\pi)}{2} - Log(b) + O(1/n),$$

where $H(p) = -pLog(p) - (1-p)Log(1-p)$ is the binary entropy function.

Notice that the communication problem is just an artifice to explain the meaning of the principle. Indeed, the best communication strategy for the communication problem corresponds to the best strategy to set up a decision tree for the given set of data according to the MDLP principle, where the complexity of the description is measured by $L(\cdot,\cdot,\cdot)$.

Our attribute selection procedure in the training phase works as follows: starting from the root, which at the beginning holds all the examples, the procedure tries to select the attribute inducing a partition on the data set which minimizes the function $L(\cdot,\cdot,\cdot)$, our complexity measure. The procedure is then applied recursively on the children of the root, and so on. It halts when there is no need for further partitions, because either the objects of the node belong to the same class or all the attributes have been selected.

We have chosen the minimum description length principle in the design of our system because it seems that this measure is *more refined* than the gain, and because it considers the *global cost* of the representation of the tree. This cost is ignored by the gain method. Experimental results presented by [18] suggested the suitability of this approach, and the performance of our prototype seem to corroborate these early intuitions.

3.2 Attributes used for Classification

As stated in the above subsections, we need to define the attributes that will be used during the training and testing phases. We report below the attributes we have used in our system. Some of them, namely C, D, G, are similar to the attributes used in [2]. Some attributes, A, B, C, and D, are evaluated on a single character of the password that is undergoing the classification, the attribute E considers pair of characters, and the last two attributes, F, and G, are evaluated on the whole password. More precisely:

- A: Returns a value in {0,1,2} when the character belongs to one of the following sets: {vowels + consonants}, {digits}, {other}.
- B: Returns a value in {0,1,2,3} when the character belongs to one of the following sets: {vowels}, {consonants} {digits}, {other}.
- C: Returns a value in {0,1,2,3} when the character belongs to one of the following sets: {vowels}, {n,m,r,l,c,g,k,x,j,q,h}, {t,d,b,p,f,v,y,w,s,z}, {digits}, {other}.

- D: Returns a value in {0,1,2,3,4,5,6} when the character belongs to one of the following sets: {vowels}, {n,m,r,l}, {c,g,k,x,j,q,h}, {t,d,b,p}, {f,v,y,w,s,z}, {digits}, {other}.
- E: This attribute considers pair of characters. Returns a value in {0,1,2,3,4,5,6} when the pair of characters belong to one of the following sets (here we use the notation v=vowel, c=consonant, d=digit, o=other, n=null): {vn, cn, dn, on}, {vv, vc, cv, cc}, {vd, dv}, {vo, ov}, {cd, dc}, {co, oc}, {dd, do, od, oo}.
- F_i (with $i = 6, 7, 8$): Returns 0 if the words is shorter then i characters, 1 otherwise.
- G_i (with $i = 1, \ldots, 8$): Returns 1 if the words contains at least i non-alphanumeric characters, 0 otherwise.

Notice that we need to decide which classification criterion must be applied when testing an attribute on a position p on a password whose length is less than p. We have identified two possible strategies: classify the character as a vowel, i.e. as belonging to the class with lowest "power", or classify it as belonging to a "special class", i.e., not belonging to any of the previously defined classes. The experiments have shown that the first solution is more suitable since we obtain higher compression ratios and lower error rates.

4 Tests and Comparisons

In this section we report the results of the experiments we have run to test the performance of our system. We start by describing the results obtained using, during the training and the testing phase, only *the first eight characters* of the words/passwords belonging to the dictionaries. In this case, we assume that the system establishes an upper bound on the length of the password the users can choose. We also report the experiments we have done implementing *dynamic* attributes, i.e., allowing the system administrator to decide how many characters of the passwords the checker has to use.

4.1 Testing the Predictive Power

To check the predictive power achieved by the trees constructed by HYPPOCRATES, we have run several training phases, followed by testing operations on dictionaries not used during the construction of the trees. Since we are interested in testing the predicting power of the tree, we do not use the dictionaries of exceptions generated by the training phase. Notice, however, that the check on the dictionaries of exceptions simply decreases the errors in case there are misclassified words that are common to the weak[6] dictionaries, used for the training, and weak dictionaries, used during the testing phase.

4.2 The Dictionaries

The dictionaries we have used for the training and the testing phases are the following: we have used for the training set a big dictionary of weak passwords, we call `Total.clean`, consisting of 1,668,118 *different* words taken from various languages, e.g., English, Italian,

[6]Notice that, we store in the file of exceptions only *weak passwords classified as strong ones*, since we consider this kind of error more dangerous than misclassify a strong password as a weak one.

Spanish, French, and some thematic dictionaries, e.g., computer, science, sport, cinema. The size of this weak dictionary approaches to 17 Mbytes.

We have used two different types of strong dictionaries. The first type, we simply call `Strong`, is composed by words that have been randomly generated using the characters reported in Table 2.

<table>
<tr><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td><td>G</td><td>H</td><td>I</td><td>J</td><td>K</td><td>L</td><td>M</td><td>N</td><td>O</td></tr>
<tr><td>P</td><td>Q</td><td>R</td><td>S</td><td>T</td><td>U</td><td>V</td><td>W</td><td>X</td><td>Y</td><td>Z</td><td>a</td><td>b</td><td>c</td><td>d</td></tr>
<tr><td>e</td><td>f</td><td>g</td><td>h</td><td>i</td><td>j</td><td>k</td><td>l</td><td>m</td><td>n</td><td>o</td><td>p</td><td>q</td><td>r</td><td>s</td></tr>
<tr><td>t</td><td>u</td><td>v</td><td>w</td><td>x</td><td>y</td><td>z</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td>8</td><td>9</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td colspan="15">Strong Characters</td></tr>
<tr><td>!</td><td>”</td><td>#</td><td>$</td><td>%</td><td>&</td><td>(</td><td>)</td><td>*</td><td>+</td><td>,</td><td>-</td><td>.</td><td>/</td><td>:</td></tr>
<tr><td>=</td><td>?</td><td>{</td><td>}</td><td>]</td><td>^</td><td>[</td><td>’</td><td>‘</td><td>@</td><td>;</td><td><</td><td>></td><td>|</td><td>~</td></tr>
<tr><td>\</td><td></td><td></td><td></td><td></td><td></td><td></td><td>%</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Table 2: The Charset

The second one, we call `Very.Strong`, is composed by randomly generated words, with at least two *strong characters* (see Table 2 for the list of strong characters). We discuss the size of these dictionaries in the next sections. We have also used noise dictionaries generated by inserting or substituting one randomly chosen letter in the weak password with a strong character. We report the results of the test ran over four weak testing dictionaries, we call `Dic.1, Dic.2, Dic.3, Dic.4`. These dictionaries contain almost 30,000 different words each, taken from four books. From these dictionaries we have constructed noisy testing dictionaries `Dic.i.1, Dic.i.2, Dic.i.3, Dic.i.4` and `Dic.s.1, Dic.s.2, Dic.s.3, Dic.s.4` by inserting, respectively substituting, one character with a strong one.

The strong testing dictionaries, `Dic.Strong, Dic.Vstrong` have been constructed using the same criterion described above for the `Strong` and `Very.Strong` dictionaries used during the training phase.

4.3 Fixed Length Attributes

In this subsection we describe the experiments done using the first eight characters of the password for the classification process. The choice of this length is due to the fact that Unix operating systems allow passwords at most eight characters long.

4.3.1 Optimizing the Compression Ratio

The first experiment we have run is directed to study the dependence of the compression ratios, which will be defined below, and the error rates with respect to the size of the strong dictionary. As we have seen in the previous sections, we need a dictionary of weak passwords and a dictionary of good ones, and the size of the training set is the sum of the size of both these dictionaries.

It is likely to think that the smaller is the training set, the smaller is the tree (trained and pruned) that classifies the training set. On the other hand, the smaller is the training set, the higher is the error occurring when the tree is used to classify passwords that do not belong to the training set.

Recall that from the point of view of password security, it is important that the checker does not accept weak passwords, while rejecting a strong password simply translates into a request for the user to type a new password in. Thus, since we want to reduce the training set, we cannot reduce the size of the weak dictionary, as all these passwords will be correctly classified as weak by the checker, but we should simply reduce the size of the strong dictionary.

Based on this idea, we have ran several experiments using `Total.clean` as weak dictionary and a randomly generated strong dictionary whose size ranges between the 5% and the 100% of the size of the weak dictionary. The results of these experiments confirm that if the size of the strong dictionary increases then also the size of the decision tree increases. We have used as strong training dictionaries both `Strong` and `Very.Strong`. All the experimental results show that using the second dictionary leads to higher compression ratios and lower error rates.

In Table 3 we report the compression ratios obtained using this approach to construct the training set for the experiments described in Section 4.3.3.

Exp. no.	Strong dic. %	C1	PC1	C2	PC2
3.1	5	1.060616%	0.030250%	1.066092%	0.248871%
3.2	10	1.500487%	0.033102%	1.517905%	0.571697%
3.3	20	2.094764%	0.049583%	2.138685%	1.077475%
3.4	30	2.520016%	0.053936%	2.590119%	1.364063%
3.5	40	2.871475%	0.059337%	2.963804%	1.781389%
3.6	50	3.166717%	0.057475%	3.282923%	2.212565%
3.7	60	3.413457%	0.058452%	3.549247%	2.417352%
3.8	70	3.642529%	0.058918%	3.795033%	2.745247%
3.9	80	3.820715%	0.062247%	3.995641%	2.975094%
3.10	90	4.020887%	0.064423%	4.209734%	3.227532%
3.11	100	4.206417%	0.063667%	4.410313%	3.541192%

Table 3: Compression Ratios

In this table, the coefficients $C1$ (resp., $PC1$), are computed as the ratio between the size of the trained (resp., pruned) tree and the size of the weak dictionary, while the coefficients $C2$, (resp., $PC2$) are computed as the ratio between the size of the trained (resp., pruned) tree and the sum of the sizes of the weak dictionary and the dictionary of exceptions. This means that the lower is the value of the coefficient, the higher is the compression ratio achieved by the experiment.

Surprisingly, the error rate obtained in these experiments show that if the size of the strong dictionary decreases, the error obtained on the weak and noise testing dictionaries decreases, too. This means that we can obtain lower error by using smaller trees. On the other hand, the error rate decreases for the strong testing dictionaries if the size of the strong dictionary used for the training increases. However, while the error rate on the weak dictionaries is acceptable, the error on the noise and strong dictionaries is often greater than 10%, that is, of course, unacceptable.

4.3.2 Decreasing the Error on Noise Dictionaries

One of the basic problem with the training strategy presented in Section 4.3.1 is that the error rate obtained on the noise dictionaries is too high. In order to decrease this error, we have ran some experiments in which we substitute the weak dictionary in the training phase, with a *noisy* one. More precisely, we construct two dictionaries `Total.clean.ins` and `Total.clean.sub`, obtained from `Total.clean` by inserting, resp., substituting, a strong character in a random position within each password of the dictionary. The results of these experiments show that the error rate obtained on noise dictionaries is close to zero.

However these kind of trainings have three side effects: first, the size of the trees obtained is greater than before. The reason for this is that the "distance" between the strong and the weak training dictionaries is "smaller" and, thus, more attributes are needed to correctly classify the training set. Second, the error rate on the weak and strong dictionaries is too high. Third, if the tree has been constructed using a noise dictionary obtained by inserting (resp., substituting) a strong character, this tree has lower error on testing dictionaries constructed using the same type of noise, i.e., insertion (resp., substitution), but has higher error on testing dictionaries obtained by substituting (resp., inserting) strong characters.

4.3.3 Decreasing Error by Mixing

The main problems with the experiments presented in Section 4.3.2 are the high error rate on strong and weak dictionaries and the dependence of the error on different noisy dictionaries. To solve these problems, we have constructed a new dictionary `Total.clean.all` obtained from `Total.clean` as follows: each word in the original dictionary has the same probability to be put in the new dictionary unchanged, with a strong character substituted, with a strong character inserted in a random position. As we will see in the next section, this strategy leads to good compression ratios and low error rates.

4.4 Comparing Hyppocrates with Other Checkers

We have compared the performance of Hyppocrates, using the results obtained by training the decision tree using the dictionary described in Section 4.3.3, with respect to two password checkers, Crack and ProCheck presented, respectively in [13] and [2]. We have used `Total.clean` to "train" Crack, as it leads to lower error rate, and we have used the best compression ratios reported in [2]. In Table 4 we report the compression ratios for each password checker.

The first column of the table contains the checker used for the testing, the second column contains the size of the training dictionary, the third column contains the size of the data structure representing the training dictionaries. The columns labelled $C1, C2, PC1$ and $PC2$ contain the compression ratios defined above. In the last column of this table we report the time required for the training phase. We notice this column is incomplete because the time required for the training of ProCheck are not reported in the paper. Moreover this last checker cannot be trained. It is provided with both a trained and a pruned tree.

In Tables 5, 6, 7, 8, we report the error rates obtained by classifying the testing dictionaries described in Section 4.2. The weak training dictionary used for these experiments is the `Total.clean.all`. On the other hand, the strong training dictionary used for each experiment is a `Very.Strong` type whose size ranges from 5% to 100%, as reported in Table 3.

Checker	training dic.(byte)	Data (byte)	C1	Exception (byte)	C2	time (sec)
3.2 Trained	17183224	257832	1.50049%	2993	1.51790%	5774
3.6 Trained	17183224	544144	3.16672%	19968	3.28292%	8878
ProCheck	1305234	32121	2.46094%	6548	2.96261%	?
Crack	16626766	7867297	47.31706%	0	47.31706%	121
Checker	training dic.(byte)	Data (byte)	PC1	Exception	PC2	time (sec)
3.2 Pruned	17183224	5688	0.03310%	92548	0.57170%	5825
3.6 Pruned	17183224	9876	0.05747%	370314	2.21256%	8949

Table 4: Comparisons with Other Password Checkers.

From the results it is immediate that Hyppocrates achieves better compression ratios and lower error rates compared to the other checkers. Recall that, in [2], the authors compared ProCheck with many other password checkers, showing that their system was significantly better than the other considered checkers.

In Table 5 we report the results of the tests using `Dic.Strong` and `Dic.VStrong` as testing dictionaries. Recall that the size of the strong dictionary used for the training is 10% (resp., 50%) of the size of the training weak dictionary in experiment 3.2 (resp., 3.6). We stress again that the bigger is the strong training dictionary, the lower is the error rate on testing strong dictionaries. Moreover, the reason for which the error rates obtained on testing `Dic.Strong` are higher than the ones obtained on `Dic.VStrong` is that we used `Very.Strong` for the training phase, and thus the "concept of good password" learned by the tree is stronger of the one proposed by this testing dictionary. Notice that the error rates on `Dic.Strong` in both the experiments 3.2 and 3.6 using the trained tree for the testing phase are almost the same. This means that almost 14% of the words in this dictionary have, with high probability, a single strong character or no strong character at all. Recall that this is possible since words in `Dic.Strong` are constructed by randomly choosing characters from the whole charset reported in Table 2. However, the results obtained by both the experiments show that using appropriate parameters for the training phase, the error rate obtained on the strong dictionaries can be much lower of the results obtained by the other checkers.

Checker	Error (%)	
	Dic.Strong	Dic.VStrong
3.2 Trained	13.8097 %	7.5911 %
3.2 Pruned	25.4671 %	15.5040 %
3.6 Trained	13.5496 %	3.1139 %
3.6 Pruned	14.2640 %	4.4056 %
ProCheck	21.7712 %	20.1322 %
Crack	13.6357 %	13.6671 %

Table 5: Testing Strong Dictionaries

In Table 6 we report the results of the tests using the weak dictionaries `Dic.1`, `Dic.2`,

`Dic.3` and `Dic.4`. First of all we notice that the error rates obtained by HYPPOCRATES in all the experiments are lower, most of the time one order of magnitude lower, than the ones obtained by ProCheck. It is also clear from the results reported that the lower is the size of the strong training dictionary, the lower is the error rate obtained on weak testing dictionaries. It is clear from the data reported in the table that decreasing the size of the strong dictionaries during the training phase results in lower error rates.
These results also show that Crack has a limited predictive power. Indeed, the error rates with this "checker" are really high.

Checker	Error (%)			
	Dic.1	Dic.2	Dic.3	Dic.4
3.2 Trained	0.0586%	0.0056%	0.1759%	0.3387%
3.2 Pruned	0.0586%	0.0000%	0.1971%	0.2909%
3.6 Trained	0.1298%	0.0641%	0.4367%	0.5254%
3.6 Pruned	0.1340%	0.0362%	0.3427%	0.5210%
ProCheck	1.5410%	0.9250%	2.5202%	1.4805%
Crack	15.0042%	19.5954%	17.1898%	17.2325%

Table 6: Testing Weak Dictionaries

In Table 7, we report the results on the test ran using the dictionaries `Dic.i.1`, `Dic.i.2`, `Dic.i.3` and `Dic.i.4`, while in Table 8 the testing dictionaries are `Dic.s.1`, `Dic.s.2`, `Dic.s.3` and `Dic.s.4`. Recall that `Dic.i.j` (resp. `Dic.s.j`) is obtained by `Dic.j` by inserting (resp., substituting) a strong character in a randomly chosen position in the password.
We do not comment the performance of Crack in this case since it is clear that the error rates are not acceptable for this kind of testing dictionaries.
Also in this case, the results obtained confirm that decreasing the size of the strong dictionaries during the training phase results in lower error rates.
Although for the class of noisy dictionaries, the performance of HYPPOCRATES and ProCheck are comparable, we notice that a careful choice of the parameters for the training phase always leads our checker to have better performance, i.e., lower error rates, compared to ProCheck.

Checker	Error (%)			
	Dic.i.1	Dic.i.2	Dic.i.3	Dic.i.4
3.2 Trained	1.3735%	1.3541%	2.0107%	2.5269%
3.2 Pruned	1.3735%	0.9362%	1.2677%	2.1058%
3.6 Trained	2.7136%	2.5800%	3.5817%	4.4547%
3.6 Pruned	3.0193%	2.8642%	3.7000%	5.0191%
ProCheck	2.1315%	1.2427%	2.9084%	2.2925%
Crack	40.5193%	47.1511%	44.9004%	43.5698%

Table 7: Testing Noise Dictionaries (Insertion)

We point out the apparently strange behaviour of the performance of the trained trees, compared to the pruned trees in the experiments reported. During the testing phase of weak dictionaries, the error rates on pruned trees are lower, as expected, than the ones obtained

Checker	Error (%)			
	Dic.s.1	Dic.s.2	Dic.s.3	Dic.s.4
3.2 Trained	1.5704%	1.3736%	2.3170%	2.5052%
3.2 Pruned	1.1307%	1.0978%	1.5952%	2.1883%
3.6 Trained	3.1826%	2.8085%	3.9002%	4.4764%
3.6 Pruned	3.1868%	2.7862%	3.7455%	4.7239%
ProCheck	3.7898%	2.6859%	4.8282%	3.7774%
Crack	47.6424%	51.1047%	48.2577%	47.7727%

Table 8: Testing Noise Dictionaries (Substitution)

by using the trained trees. The situation is reversed when testing the strong dictionaries: in other words, the error rates associated with the pruned trees are higher than the ones associated with the trained trees. The situation is more complicated when testing the noisy dictionaries. The error rates given by trained and pruned trees are roughly the same.

This behaviour can be explained as follows. The pruning operation induces an error on the classification of the training set, but it is supposed to enhance the predictive power of the tree. However, when the size of the strong dictionary in the training set is small (i.e. 10% of the weak dictionary, experiment 3.2), the classification error on the testing set, induced by the pruning operation, is amplified, since the tree has had a small set of examples for learnig the concept of "strong". This is clear in the results reported in Table 5, where the gap between the error rates of the pruned and trained trees decreases as the size of the strong training dictionary increases.

On the other hand, since the size of the weak training dictionary is the same for all the experiments, for the testing phase on weak dictionaries, the error rates of the pruned tree is always lower that the one for the trained trees.

Considering the results obtained in the experiments just described, we believe that using a small strong training dictionary (compared to the weak one) has both the advantage of generating a small tree and to lead to lower error rates for weak and noisy passwords. The main drawback of this approach is that the error rates on strong dictionaries will be high but, we think, acceptable, in order to guarantee the security of the system.

4.5 Dynamic Length Attributes

Many systems allow passwords with more than eight characters. Our package enables to select the maximum number of characters of the password to be used for the classification process. In this subsection we discuss this issue.

4.5.1 Choosing the Optimal Length

The possibility of choosing an arbitrary number of characters, implies that we need to decide which is the optimal one for a given training set. In other words, we need to decide how many characters the checker must consider during the training and the testing phase. Intuitively, this length should depend on the average length of the passwords in the training set. Indeed, if we use few characters, the decision tree should not have enough information to correctly classify the training set. On the other hand, it should be useless to check the i-th character in the password if i is much greater than the average length, since few passwords will be classified by means of that character.

To argument this idea, we have run several experiments whose description follows: the training set consists of the dictionary `Total.clean.all` described in Section 4.2, whose average password length is 9.3. We have constructed strong dictionaries, with size ranging from 10% to 100% of the size of the weak dictionary, with the same average password length. We allow the system to use the first $6, 8, 12, 16$ characters of the password for the classification.

The results of these experiments show that, if we look at experiments that use the same number of characters for the classification, as the size of the strong dictionary increases, the size of the decision tree increases too. Moreover, the error rate on weak and noise dictionary increases, while, as before, the error rate on the strong dictionary decreases. It is interesting to see that the lowest error rates are achieved when the number of characters used for classification approaches to the average length of the passwords in the dictionary.

5 Conclusions

In this paper we have described a new realization of a proactive password checker. We have explored the possibility of applying the Minimum Description Length Principle, a key idea of Machine Learning, in the context of proactive password checker. The results reported in the tests and comparisons section of our work indicate that the pursued approach is suitable. HYPPOCRATES has been tested on a 450 MHz Pentium III Linux box with 8Gb EIDE hard disk. The code of the prototype implementation has been written using the ANSI C language and the graphical interface has been realized for X-Windows system. We are working on patches for the unix-like *passwd* command. A beta version of the software will be available from our Web site (*http://www.security.unisa.it/PPC*).

References

[1] Anderson., J., 1980. Computer security threat monitoring and surveillance. Tech. rep., Fort Washington, PA.

[2] Bergadano, F., Crispo, B., Ruffo, G., 1998. High dictionary compression for proactive password checking. ACM Transactions on Information and System Security 1, 3–25.

[3] Bishop, M., 1991. Password management. In: Proceedings of COMPCON 1991. pp. 167–169.

[4] Bishop, M., 1992. Anatomy of a proactive password checker. In: Proceedings of the Third UNIX Security Symposium. pp. 130–139.

[5] Bishop, M., 1992. Proactive password checking. In: Proceedings of the Fourth Workshop on Computer Security Incident Handling. pp. W11:1–9.

[6] Bishop, M., 1995. Improving system security via proactive password checking. Computers and Security 14 (3), 233–249.

[7] Bloom, B., 1970. Space/time trade-offs in hash coding with allowable errors. Communications of ACM .

[8] Boyko, V., MacKenzie, P., Patel, S., 1991. Provably secure password-authenticated key exchange using diffie-hellman. In: Proceedings of Eurocrypt 2000. Vol. LNCS vol. 1807. pp. 156–171.

[9] Breiman, L., Friedman, J. H., Olshen, R. A., Stone, C. J., 1984. Classification and Regression Trees. Wadsworth & Brooks/Cole, Advanced Books & Software, Pacific Grove, CA.

[10] Davies, C., Ganesan, R., 1993. Bapasswd: A new proactive password checker. In: Proceedings of the 16th National Conference on Computer Security. pp. 1–15.

[11] Katz, J., Ostrovsky, R., Yung, M., 2001. Efficient password-authenticated key exchange using human-memorable passwords. In: Proceedings of Eurocrypt 2001. Vol. LNCS vol. 2045. pp. 475–495.

[12] Klein, D. V., Summer 1990. Foiling the cracker – a survey of, and improvements to, password security. In: Proceedings of the second USENIX Workshop on Security. pp. 5–14.

[13] Muffett, A. D., 1992. Crack 5.0.

[14] Nagle, J. B., 1988. An obvious password detector.

[15] Neuman, B. C., Tso, T., 1994. Kerberos: An authentication service for computer networks. IEEE Transactions on Commun. 32, 33–38.

[16] Quinlan, J. R., 1987. Simplifying decision trees. Int. Journal of Man Machine Studies 27, 221–234.

[17] Quinlan, J. R., 1992. C4.5: Program for Machine Learning. Morgan Kaufmann Publishers Inc., San Francisco, CA.

[18] Quinlan, J. R., Rivest, R. L., 1989. Inferring decision trees using the minimum description length principle. Information and Computation 80 (3), 227–248.

[19] Rissanen, J., 1986. Stochastic complexity and modeling. The Annals of Statistics 14 (3), 1080–1100.

[20] Spafford, E., 1992. Opus: Preventing weak password choices. Computers and Security 3.

[21] Stalling, R., 1995. Network and Internetwork Security Principles and Practice. Prentice Hall, Englewood Cliffs, New Jersey.