A Novel Approach to Proactive Password Checking

Carlo Blundo[1], Paolo D'Arco[2], Alfredo De Santis[1], and Clemente Galdi[3]

[1] Dipartimento di Informatica ed Applicazioni
Università di Salerno, 84081, Baronissi (SA), ITALY
{carblu,ads}@dia.unisa.it
[2] Department of Combinatorics and Optimization
University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada
pdarco@cacr.math.uwaterloo.ca
[3] Computer Technology Institute and
Dept. of Computer Engineering and Informatics
University of Patras, 26500, Rio, Greece
clegal@ceid.upatras.gr

Abstract. In this paper we propose a novel approach to strength password-based access control strategies. We describe a proactive password checker which uses *a perceptron* to decide whether a user's password is easy-to-guess. The checker is simple and efficient, and it works since easy and hard-to-guess passwords seem to be *linearly separable.* Experimental results show that the error rates in many cases are close to zero, memory requirements can be quantified in few bytes, and the answers to classification queries are almost immediate. This research opens new directions to investigate on the applicability of neural network techniques to data security environments.

Keywords: Data Security, Access Control, Proactive Password Checking, Perceptron, Neural Network.

1 Introduction

Authentication and identification protocols to check users' accesses to partially shared or private resources are deep problems for computer scientists involved in studies on data security. Several methods have been proposed in the recent years but, among these, password-based access control strategies are still frequently used for their simplicity.

A user, who wants to identify himself to a system, engages an interactive identification protocol: First, the user types in his login name; then, the system asks for the *password*, a secret word which proves the real identity of the user. The system stores in a database a reference string encrypted using the password as a key. So, when the user types in his password, the system re-encrypts the reference string and compares it with the stored one. If they match, the access is allowed; otherwise, it is refused. The scheme is supposed to be secure if the user keeps secret his password.

Security of Password-based Systems. Illegal logging onto a machine often happens by mean of users' passwords. This is possible not only when the user accidentally discloses his password but even when it is easy-to-guess [8]. Indeed, if an hacker can obtain in some way the file containing the reference string encrypted with the users' passwords, used by the system to allow and refuse the access, then he can try off-line to encrypt the reference string with all the words of a dictionary until a match is found. This attack is called *exhaustive search.* Excellent softwares to accomplish this task have been developed in the recent years, and are available on-line (see, for example, [9]).

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.

Previous works. The problem of using hard-to-guess passwords has been studied in several papers. So far, four techniques [3, 4] have been proposed to eliminate easy-to-guess passwords, and the more promising seems to be the *Proactive Password Checking* approach.

A proactive password checker conceptually is a simple program. It holds a list of easy-to-guess 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 refused and a short justification is given; otherwise, the substitution is allowed. The philosophy these programs are based on is that the user has the possibility to select a password but the system enables the change only if it is a "non trivial" one. However, a straightforward implementation of such a program is not suitable since the list can be very long and the time to check membership can be high.

Various proactive password checkers that aim to reduce the *time* and *space* requirements of this trivial approach have been proposed [10, 11, 5]. All these models are an improvement upon the basic scheme. An interesting approach to design a proactive password checker is the one applied in [1], and subsequently improved in [6], where the problem of password classification is viewed as a *Machine Learning Problem.* The system, in a training phase, using dictionaries of examples of easy and hard-to-guess passwords, *acquires the knowledge* to distinguish between them. This knowledge is represented by a *decision tree* that is used afterwards by the checker to accept or refuse a password change. The experimental results reported in [1, 6] show a meaningful enhancement on the error rates with respect to the previous solutions.

In this paper we introduce a new idea to proactive password checking: We use a *perceptron* to distinguish easy-to-guess passwords from hard ones. During the training phase, the perceptron learns the differences between "easy" and "hard" by means of examples. Then, the perceptron is used in the testing phase to decide on the membership of a password to one of the two classes. The technique is promising since the checker shows small error rates, and requires very low memory storage (in our prototype, only 40 bytes!). It is interesting to point out that basically the perceptron codes some rules to classify passwords. Other checker which implements a direct approach (i.e., a code checking with a series of *if-then-else* statements a bunch of rules) requires non negligible time and memory requirements. We obtain better performances evaluating a simple weighted sum and using only 40 bytes. Hence, this approach is suitable for implementations even on devices with very poor hardware facilities (i.e.,smart cards).

Easy and Hard-to-Guess. We have informally referred to easy and hard-to-guess passwords. We define *easy-to-guess* as a condition of membership in some "easy to exhaustively search" dictionary, and *hard* by negation of easy. Notice that these notions are *computational in nature*. Hence, a password is easy if it is "guessable" in reasonable time, while it is hard if the guessing requires unavailable resources of time and space. Usually, a hard-to-guess password looks like a random string on the reference alphabet.

2 Mathematical Framework

The above notions can be modeled in a mathematical setting. In this section we briefly describe the problem of *choosing* a password p from a set $\mathcal{P}$. More precisely, along the same line of [3], we characterize the selection of *easy-to-guess* passwords from $\mathcal{P}$ in terms of certain probability distributions.

Let $\mathcal{P}$ be the set of all admissible passwords, let p be an element chosen from $\mathcal{P}$, and let s be the function used to select the password p from $\mathcal{P}$. Then, denote by p' a *guess* for the password p and assume that it takes a constant amount of time $T = t(p')$ to determine whether this guess is a correct one.

We can model the *choice* of p in $\mathcal{P}$ with a random variable $\mathbf{S}$, taking values in $\mathcal{P}$. These values are assumed according to a probability distribution P_S upon elements of $\mathcal{P}$ that is induced by the selection function s. Moreover, the *time to guess* p can be represented with a random variable $\mathbf{F}_{P_S}$, which takes values in R^+ according to P_S.

If $\mathbf{S}$ is uniformly distributed on $\mathcal{P}$, i.e., $P_S{=}U$, and *no prior knowledge* of the authentication function (the function used by the operating system to check the equality of a guess with the true password) is available then, as pointed out in [3], to guess the selected password p, we have to try on average $\frac{|\mathcal{P}|}{2}$ passwords from $\mathcal{P}$, and the expected running time is

$$E(\mathbf{F}_U) = \sum_{i=1}^{\frac{|\mathcal{P}|}{2}} T = T\frac{|\mathcal{P}|}{2}. \tag{1}$$

In this model, there is a correspondence between the set S of the selection functions and the set $D_{\mathcal{P}}$, the set of all probability distributions on the set $\mathcal{P}$. Indeed, we can characterize the bad selection functions s to choose p in $\mathcal{P}$, with those probability distributions P_S such that

$$E(\mathbf{F}_{P_S}) \leq kE(\mathbf{F}_U). \tag{2}$$

The parameter $k \in [0, 1]$ defines a lower bound to the acceptability of a given selection function, represented by the distribution P_S. If p is chosen according to a probability distribution P_S that satisfies (2), we say that p is *easily guessable*.

A family of bad selection functions is represented by *language dictionaries*, where the dictionary can be seen as the image-set of a selection function s. The words in the dictionary are a small subset of all the strings that can be constructed with the symbols of a given alphabet. According to our model, the distribution induced by natural languages are *skewed* on $\mathcal{P}$, since they assign non zero values only to a small subset of elements, therefore $E(\mathbf{F}_{P_S})$ is much smaller then $E(\mathbf{F}_U)$. Hence, it is sufficient to try a number of password smaller than $\frac{|\mathcal{P}|}{2}$ to guess the chosen p.

To assure the security of the system against illegal accesses we have to require that the selection function does not localize a small subset of $\mathcal{P}$. This means that we have to find a method *to discard* those probability distributions P_S on $\mathcal{P}$ such that $E(\mathbf{F}_{P_S})$ is too much small. If $E(\mathbf{F}_U)$ is very large and we can force P_S to look like U, then the goal is obtained.

A proactive password checker can be viewed as a tool to guarantee that a password p is chosen from $\mathcal{P}$ according to a suitable distribution, i.e., a distribution that looks like the uniform one. It works like a *sieve* on the set $D_{\mathcal{P}}$. Actually, the proactive checker does not distinguish the different distributions but simply distinguishes among good distributions, close to the uniform one, and bad ones, close to the distributions induced by natural languages.

This is a general analysis of the password choosing problem. In order to derive practical results we need to carefully specify the password space $\mathcal{P}$. In our setting this set is the set of all strings of length less than or equal to 8, composed by "printable" ASCII characters. This set is reported in Section 5.

3 Pattern Recognition

Pattern recognition concerns with objects categorization [2]. It is a solid area of studies in Artificial Intelligence. The objects of a given universe can belong to different classes, according to their own characteristics. The recognition problem consists in associating each object to a class.

A pattern recognition system can be seen as a two-stage device: A feature extractor and a classifier. It takes as input an object and outputs the classification.

A *feature* is a measurement taken on the input object that has to be classified. The values of the measurements are usually real numbers and are arranged in a vector called *feature vector*. The set of possible feature vectors is called *feature space*. The feature extractor of a pattern recognition system simply takes measurements on the object and passes the feature vector to the classifier. The classificator applies a given criterion to establish in which class the object does belong to.

Discriminant functions are the basis for most of the majority of pattern recognition techniques. A *discriminant function* is a function that maps the feature vector onto the classification space, and usually defines a boundary among the classes. If the discriminant function is a linear function, i.e., it defines a boundary in the classification space that looks like an hyperplane, the classifier is said linear. Of course, a linear classifier can be used if the classes themselves can be separated by means of a straight line. When this happens, we say that the problem is linearly separable.

As we will see later, the system we are looking for is a pattern recognition system which takes as input words, extracts some features from them, and then outputs a decision on their membership to the easy-to-guess class or the hard one. To classify, our device uses a perceptron which realizes a linear classifier.

Neural Computing: The Perceptron Neural computing is an alternative way to do computation. Against the traditional approach of computer science, a neural machine learns solving problems by trials and errors. In a training phase the machine sees a sequence of examples with the corresponding solutions and adapts its internal parameters to match the correct behaviour. When the training phase stops, the machine is ready to solve new and unseen instances of the problem.

The approach is quite interesting since the machine holds simply the software to manage the training process. The knowledge to solve the specific problem is acquired during the learning phase and is stored in the modified values of the internal parameters. Therefore, the machine is in some sense *self-programmable.*

A formal neuron is a model which tries to capture some aspects of the behaviour of the cerebral neuron. A first model was proposed in 1943 by McCulloch and Pitts. It was a simple unit, thresholding a weighted sum of its input to get an output. Frank Rosenblatt, in 1962, in his book *Principles of Neurodinamics* introduced the name *Perceptron.*
The learning rule of the Perceptron consists of the following steps

- Set the weights and the threshold randomly
- Present an input
- Calculate the actual output by taking the threshold value of the weighted sum of the inputs
- Alter the weights to reinforce correct decisions and discourage incorrect ones

This type of learning is called *hebbian* in honour to Donald Hebb, who proposed in 1949 a similar rule starting from his studies on real neural systems.

It is well known (and not difficult to see) that the Perceptron implements a linear classifier. Indeed, the weighted sum defines an hyperplane in the Cartesian space. If the weighted sum of an input is greater than the threshold, then the pattern belongs to the class on one side of the hyperplane, otherwise it is an element of the class on the other one.

Intuitively, our problem is linear separable, i.e., easy-to-guess passwords and hard ones present characteristics which permit the separation of the two classes by means of a straight line. Under this hypothesis, we have designed the kernel of the proactive password checker. The results obtained seem to confirm this intuition.

4 The Checker

The proactive password checker presented in this paper is based on a perceptron. The training process uses two dictionaries, a dictionary of *hard-to-guess* words and a dictionary of *easy* ones. From each word, chosen at random in one of these dictionaries, we extract some features and use the perceptron to classify. The learning rule used to train the perceptron is the *Widrow-Hoff delta rule* [2], a specific implementation of the general learning algorithm described before.

Thus, in order to classify the passwords, we have to identify some features that are relevant for the classification. One of the first features that should be considered is the *length* of the password. Actually, it is commonly believed that, the longer is the password, the harder is to guess. However, the length is not sufficient (and sometimes is a wrong criterium) to correctly classify hard-to-guess passwords and easy ones.

Following the *intuition*, and by *trials and errors*, we have identified four features for the classification called: *Classes, #Strong Characters, Digrams, Upper-Lower Distribution.* More precisely:

- Classes: It is reasonable to consider the set of ASCII characters divided into classes of different strength. Commonly, passwords are composed by letters, this means that all the (upper and lower case) letters must have low values. In a second class, we can put the digits `'0',...,'9'`. This is because it is not frequent to find a digit in a password, but it is not so unusual, too. In the last class, called the class of *strong* characters, we can put every character that does not belong to the first two classes. To mark the *distance* among these classes we have assigned to the class of letters a value equal to 0.2, to the class of digits a value equal to 0.4 and to the last class 0.6. The *overall value* of a password is computed by summing up the value associated to each character in the password. Notice that, since the feature is a sum, the longer is the passwords the higher is the value.
- #Strong Characters: The second feature is the *number* of strong characters contained in the password.
- Upper-Lower Distribution: The value of this feature is calculated by the following formula: $|UPP - LOW|/\ell et$, where UPP is the number of upper case letters, LOW is the number of lower case letters and ℓet is the number of letters in the password. The presence of this feature is due to the observation that passwords that contain both upper and lower case letters are lightly stronger that passwords composed by lower (upper) case letters only.
- Digrams: This feature looks at the *types of digrams* present into the password. More precisely, we say that a digram is an *alternance* if the two characters of the digram belong to different classes. The checker scans the password, analyzes all the digrams from the left to the right, and assigns values to each of them. The more alternances the password has, the higher is the value.

5 Description of the Experiments

Many operating systems limit the length of the password. For example, Unix-like operating systems work with passwords of length at most 8 characters. At the same time, many others do not accept passwords with length less than 6 characters. For this reason, we have used, in the training and testing phases of our experiments, dictionaries of hard-to-guess passwords by generating random words with length ranging between 6 and 8 characters. Similarly, passwords contained in the easy-to-guess dictionaries, collected from several sources, have been truncated to the first eight characters.

The Dictionaries. We have used eight dictionaries, *strong.0, strong.1, strong.2, weak, noise.0.1, noise.0.2, noise.1.1* and *noise.2.2*, for the training phase, and nine dictionaries *test.strong.0, test.strong.1, test.strong.2, test.weak.0, test.weak.1, test.noise.0.1, test.noise.0.2, test.noise.1.1* and *test.noise.1.2*, for the testing phase. We briefly describe each of them. The dictionaries *strong.0, strong.1 strong.2* are dictionaries of *random* words. They are composed by pooling together 30000 words of length 6, 30000 words of length 7 and 30000 words of length 8.

The difference among them is the generation rule used. The *strong.0* dictionary has been generated by randomly choosing the characters of each word in the following set:

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

The random selection has been done using the `random()` C-function. This function implements a non-linear pseudorandom generator that is assured to have a long period.

Recall that, we have said before that a character is *strong* if it belongs to the following set:

```
: ; < = > ? @ [ ] ^ _ ‘ ’ { | } ! " # $ % & ( ) * + , - . / ~
```

The dictionaries *strong.1* and *strong.2* have been constructed using the same rule as the one used to construct *strong.0* with additional constraints. More precisely, each word in *strong.1* has *at least one* strong character or *at least two* characters must be digits. Similarly, each word in *strong.2* contains at least *two* strong characters or *three* digits. Intuitively, the "strength" of these dictionaries increases from *strong.0* to *strong.2*.

About the dictionaries of easy-to-guess passwords, the first one, *weak*, is composed by words having length from 1 to 18, recovered from several sources and books, truncated to the first 8 characters. All the passwords in this dictionary are composed by *lower case* letters. The dictionary *noise.0.x*, for $x = 1, 2$, is constructed from the dictionary *weak* by substituting x randomly selected characters with strong ones. The dictionary *noise.1.x*, for $x = 1, 2$, is constructed from the dictionary *noise.0.x* by substituting *half of the lower case letters* with the corresponding upper case ones.

For the testing phase we have used dictionaries presenting the same characteristics of the training dictionaries. This similarity is reflected in the name that is the same up to the prefix *test* (i.e., test.strong.0, and so on). The only difference is that we have tested two weak dictionaries, *test.weak.0* and *test.weak.1* where the first one as the same characteristics of *weak*, while *test.weak.1* is constructed from *test.weak.0* by substituting half of the lower case letters with the corresponding upper case ones.

The Intuition Behind the Experiments. The training phase is a critical step: Based on the easy and hard-to-guess examples presented, the perceptron learns different notions of easy and hard. Hence, if the training set is not accurately chosen, the perceptron can give poor performances. The first experiment we have run is to train the perceptron with a "very easy" dictionary of easy examples, and a "very hard" dictionary of hard ones. Unfortunately, the results obtained in testing phase are not exciting applying this strategy. The main problem with this approach is that there is a big "distance" between the dictionaries. So, many noisy passwords are classified as hard-to-guess.

To avoid this phenomenon, we have used dictionaries whose distance is small enough to obtain a more refined notion of "easy" and "hard" but, at the same time, big enough to ensure that the perceptron learns these distinct notions.

The Experiments. We have trained the perceptron using all the possible combinations between the strong dictionaries and the weak ones described in Section 5, *fixing* the number of examples in each training.

Since this number is smaller than the number of words in the dictionaries, the training has been *randomized.* More precisely, the training procedure, in each step, randomly chooses either the hard or the easy dictionary, and then randomly selects a word in the selected dictionary. After each training, we have tested all the testing dictionaries. Since the training is randomized, we have repeated the overall process 100 times, to ensure that the results obtained were consistent and not due to a "lucky" random sequence of examples. In Table 1 of the Appendix we report the average number of modifications of the perceptron's parameters (i.e., weights) occurred during the training phase before the convergence to a stable configuration.

Results reported in Table 1 clearly states that if the distance between the dictionaries used during the training phase is large, the process converges almost immediately. On the other hand, if the distance between these dictionaries is really small, the same process takes a while (and, maybe, does not converge).

For space limits, we report the results only for the experiments associated to the training executed using *strong.1* and *strong.2* as strong dictionaries. In Table 2 in the Appendix we report the expected error and the variance obtained on each testing dictionary.

It is immediate to see that if the pair of dictionaries approaches to the space of strong passwords, i.e., moving from *(strong.1, weak)*, to *(strong.2, noise.0.1)*, the error obtained on easy testing dictionaries decreases since the perceptron learns a stronger notion of "easy". At the same time, the error obtained on the dictionaries of hard-to-guess words increases since the perceptron also learns a stronger notion of "hard".

Platform. The experiments presented have been run on a 450 Mhz Pentium III machine running a Linux kernel 2.2.10 with 256 MBytes of RAM, and a 9 GBytes SCSI hard disk. On this machine, the software has checked more that 56,000 passwords/sec. Our tests show that this approach is significantly faster than all the previous proposed ones.

6 Conclusions and Open Problems

This paper gives more questions than answers. Following an intuition, and proceeding during the experimental phase by trials and errors, we have shown that the hard-to-guess password set (i.e., words sampled according to a uniform-like distribution on the set of printable ASCII characters) seems to be linearly separable from the easy-to-guess set (i.e., set of "structured words" chosen, for example, according to the distribution of a natural language).

Several problems arise from this study. Strictly related to the topic of this paper, the following issues could be of interest: From an experimental point of view, it would be nice to investigate other features for the perceptron, in order to obtain better performances during the classification task. The features we have used seem to be reasonable but a more accurate choice can produce a more refined classification. Along the same line, other strategies (dictionaries) for the training phase can give an improvement as well. On the other hand, from a theoretical point of view, it would be nice to prove in a formal model that easy-to-guess and hard-to-guess passwords are really linearly separable. This intuition, at the basis of the present work, seems to be corroborate by the experimental results.

Finally, we put forward the question of the applicability of neural networks to data security problems. Further investigation on the relation among these two fields could be done by researchers belonging to both fields.

References

1. F. Bergadano, B. Crispo, and G. Ruffo, *High Dictionary Compression for Proactive Password Checking*, ACM Transactions on Information and System Security, Vol. 1, No. 1, pp. 3-25, November 1998.
2. R. Beale and T. Jackson, **Neural Computing: An Introduction**, IOP Publishing Ltd, Institute of Physics, 1990.
3. M. Bishop, *Proactive Password Checking*, in Proceedings of 4th Workshop on Computer Security Incident Handling, 1992.

4. M. Bishop, *Improving System Security via Proactive Password Checking*, Computers and Security, Vol. 14, No. 3, pp. 233-249, 1995.
5. B. Bloom, *Space/Time Trade-offs in Hash Coding with Allowable Errors*, Communications of ACM, July 1970.
6. C. Blundo, P. D'Arco, A. De Santis, and C. Galdi, *Hyppocrates: A new Proactive Password Checker*, Proocedings of ISC01, Springer-Verlag, LNCS, Vol. 2200, Malaga, October 1-3, 2001.
7. C. Davies, and R. Ganesan, *Bapasswd: A new proactive password checker.* In Proceedings of the 16th National Conference on Computer Security (Baltimore, MD, Sept. 20-23).
8. D. Klein, *Foiling the Cracker: A Survey of, and Improvements to, Password Security.* Proceedings of the Fifth Data Communications Symposium, September 1977.
9. A. Muffett, *Crack 5.0*, USENET News.
10. J. B. Nagle, *An obvious password detector.* USENET News.
11. E. Spafford, *OPUS: Preventing Weak Password Choices* in Computers and Security, No. 3, 1992.

Appendix

Strong Dictionaries					
strong.0		strong.1		strong.2	
weak	102	weak	7	weak	2
noise.0.1	699	noise.0.1	385	noise.0.1	10
noise.0.2	2113	noise.0.2	1653	noise.0.2	1286
noise.1.1	3882	noise.1.1	3637	noise.1.1	10
noise.2.2	7495	noise.2.2	6980	noise.1.2	7150

Table 1. Average Number of Weight Modification over 20,000 Examples

	strong.1 weak		strong.1 noise.0.1		strong.2 weak		strong.2 noise.0.1	
Testing Dict.	Error (%)	Var. (%)	Error (%)	Var. (%)	Error (%)	Var. (%)	Error (%)	Var. (%)
test.weak.0	0.02	0.01	0.00	0.00	0.01	0.01	0.00	0.00
test.weak.1	5.36	18.09	0.02	0.00	0.01	0.01	0.00	0.01
test.noise.0.1	95.62	0.77	0.33	2.19	53.81	41.88	0.02	0.00
test.noise.0.2	99.49	0.13	79.99	0.50	91.25	8.69	79.92	0.00
test.noise.1.1	96.32	1.09	96.00	0.00	45.28	40.80	27.59	38.74
test.noise.1.2	99.59	0.12	99.56	0.00	89.70	8.51	85.91	8.13
test.strong.0	4.91	0.76	7.68	0.40	16.85	7.84	20.94	6.84
test.strong.1	0.00	0.01	1.70	0.30	11.72	8.42	16.07	7.38
test.strong.2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 2. Testing Behaviour