A Note on Proactive Password Checking

Jianxin Jeff Yan

Computer Laboratory, University of Cambridge
Jeff.Yan@cl.cam.ac.uk

ABSTRACT

Nowadays, proactive password checking algorithms are based on the philosophy of the dictionary attack, and they often fail to prevent some weak passwords with low entropy. In this paper, a new approach is proposed to deal with this new class of weak passwords by (roughly) measuring entropy. A simple example is given to exploit effective patterns to prevent low-entropy passwords as the first step of entropy-based proactive password checking.

Keywords

Proactive password checking, dictionary attack, entropy

1 INTRODUCTION

Password security is an old problem. Due to the limitation of human memory, people are inclined to choose easily guessable passwords (e.g. phone numbers, birthdays, names of family or friends, or words in human languages) that might lead to severe security problems. Though it was commonly believed that secure passwords were difficult to remember and easy-to-remember passwords were insecure, a recent experiment [14] showed with hard data that passwords based on mnemonic phrases could provide both good memorability and security, but non-compliance with password selection advices was a main threat to password security.

Proactive password checking has been a common means to enforce password policies and prevent users from choosing easily guessable passwords in the first place. When a user chooses a password, a proactive checker will determine whether his password choice is acceptable or not, and this proactive checking is done online and the user will be immediately responded the result. Among common approaches to improving password security by selecting good passwords, such as user education, program-controlled password generation and reactive password checking (i.e., system administrators periodically run password cracking programs to search weak passwords), proactive password checking has been widely regarded as the best [2, 3, 6, 11].

In 1999, Wu [13] reported a password experiment done in a Kerberos setting where a proactive password checker was used. Wu recommended strong password authentication protocols such as EKE [1], SRP [12] as an alternative approach to improving password security, since the checker appeared not to help that much in his experiment but those password authentication protocols appeared to eliminate the threat of offline dictionary attack on passwords. As far as we know, this is the most negative criticism of proactive password checking so far. Wu's claim of inefficacy of proactive password checking, however, is unconvincing, since the poor experimental result shown in his paper might and **does** simply suggest that the proactive checker used in his experiment was not effective.

On the other hand, although strong password authentication protocols utilize cryptographic technologies to generate strong session keys from passwords so that users may use weak passwords in some circumstances, they are not the "silver bullet" for password security. Firstly, these methods are expensive, and are not widely deployed in fielded systems. It also appears impossible to apply this technique in every place where the password mechanism is needed. Secondly, they are complex and error-prone. Security protocols are notoriously difficult to be correct, and subtle security flaws of protocols have been published from time to time. There are already some attacks against password authentication protocols, e.g. [10], published in the literature. Moreover, nobody is sure whether those authentication protocols are secure as claimed or not before they are rigorously proved. Most importantly, the party that stores the password file can still do offline dictionary attack as usual. Although strong password authentication protocols do amplify the search space of a weak password to that of a much stronger cryptographic key, this amplification is meaningful only when an attacker does dictionary attack against the key. The threat of offline dictionary attack, launched directly on the password file, is still there. For example, there might be an internal attack. Moreover, password authentication protocols are vulnerable to either online dictionary attack or denial of service attack. If these protocols are designed to be resilient to the

former attack, then they are unavoidably vulnerable to the latter attack, and vice versa.

Password security is not a problem that can be solved only by technical means. Human factor is also very important[1]. In case users become complacent due to the (superficial) technical advantage of strong password authentication protocols - the advertisement in many papers is that a user can use "weak", "memorable" passwords - so that they choose their names or user IDs, phone numbers or whatever is simple as their passwords, it is not impossible for an attacker to easily guess them after a few tries in an online dictionary attack.

Therefore, in places where strong password authentication does or does not fit, good password selection policies enforced by proactive checking is still one of desirable methods to improve password security in practice. From Wu's experiment, we also see a great and urgent demand of effective proactive password checking tools in real life. Nevertheless, proactive password checking is not a perfect technique. In this paper, we try to address a common shortcoming of the state of the art of proactive checking and propose a remedy.

2 PROACTIVE PASSWORD CHECKING: STATE OF THE ART

Theoretically, it is easy for proactive password checking to block all possible weak passwords. It might, for example, be enough to enforce such a simple password policy: each password has no less than eight characters, among which there is at least one lower case character, at least one uppercase character, at least one numerical character and at least one punctuation character, and there is no character occurring more than twice. Nevertheless, it is impractical to do that in real life, since passwords complying with such a policy might be too difficult to be memorized. There are always some trade-offs between security and user convenience for password choice. As a basic assumption, proactive password checking algorithms typically do not enforce extremely strict policies but allow users to choose "good enough" passwords, though the criteria of "good enough" might vary in different circumstances.

2.1 Dictionary Attack: the Basis of Current Proactive Password Checking

When a hacker cracks passwords, he can use the following two methods: 1) to do a dictionary attack, which tries each of a list of word and other possible weak passwords, and simple transformations such as capitalizing, prefixing, suffixing or reversing a word as a candidate until the hashed value of the candidate matches a password hash; and 2) to launch a brute force attack to search the whole key space, which is commonly huge. Hackers, however, always prefer to use dictionary attack, because it has proved to be very effective in history [6]. Following a similar thinking, current proactive password checkers are based on the dictionary attack. They check each user-chosen password candidate against a dictionary of weak passwords. If a candidate matches a dictionary item, or anyone of its variants that are generated by common transformations, then the candidate is an unacceptable password and rejected.

For example, *crack* [7] is one of the most popular password-cracking software. Although it also supports the brute force attack, it has been far more used as a dictionary-based cracking tool. Utilizing a similar dictionary-based algorithm used by *crack*, its designer also implemented *cracklib* [8], a proactive password-checking library, which has been integrated into some password systems. Armed with a same dictionary, a *cracklib*-supported system can prevent all weak passwords that can be guessed by *crack*.

In order to cover as many weak passwords as possible, a huge dictionary is commonly required for this dictionary-based proactive password checking. The dictionary file may occupy tens of megabytes or even more storage space, and it may take a very long time to search the huge dictionary. Consequently, an essential research problem has been how to efficiently store and search the dictionary. The search speed is even a more important concern than storage space, because proactive password checking needs to be done online in real-time while a user waits for an immediate response from the system.

The *cracklib* v2.7 included a dictionary of 1.4 million words, which had a raw size of around 15MB, and the whole package of all files occupied around 7MB, which was around 45% of original size. It used a modified-DAWG (Directed Acyclic Word Graph) compression algorithm, which preprocessed sorted lists of words to remove redundancy and make compression tools like *gzip* more effective. A gzipped-DAWG dictionary was typically about 50% of the size of the gzipped non-DAWGed dictionary [8]. In order to improve search speed, *cracklib* used an index file to access dictionary words, and kept a table to assist binary searching. In summary, the algorithms used by *cracklib* to optimize dictionary storage and checking speed were very intuitive, and they only worked efficiently when the dictionary file was of a modest size.

Researchers have been looking for good algorithms that could achieve both fast checking speed and effective dictionary compression at the same time. For example, Spafford used Bloom filters [4] in his *OPUS* system [11]. Davies and Ganesan used trigrams and a Markov model in their *BApasswd* [5]. The state-of-the-art of proactive password checking is *ProCheck* [2, 3], which uses decision-tree techniques to achieve high dictionary compression (up to 1000:1) as well as a fast checking speed. In its current implementation, a decision tree classifier with the size of only 24 KB is generated from a 28 MB dictionary file of 3,215,846 words. The proactive checking

[1] When implementing proactive password checking, it is also crucial to consider human factor, organizational or social issues. For example, users should be given good feedback on why passwords fail the test, and how they can choose a better one (now, they can be instructed to choose mnemonic phrase based passwords as suggested by [14]). In case passwords are rejected without appropriate explanation and further instruction, users may dislike the system and try to undermine it. The detailed discussion of this is, however, beyond the scope of this paper.

algorithm only searches the small classifier to determine whether a password is acceptable or not. As far as we know, *ProCheck* provides the fastest checking speed and best compression of a huge dictionary.

When a dictionary used by a proactive password checker does not match that used by a cracker, it is likely that the checker will fail to prevent some weak passwords, which can be successfully guessed by the cracker afterwards however. Although this appears to be an inherent difficult-to-solve defect for password checking, the *ProCheck* technique makes it possible for security defenders to arm themselves with dictionaries as huge as they like, and thus significantly minimize the chance window of bad guys.

2.2 A Common Shortcoming

Even though a word in a live language is extremely difficult to memorize, or seldom used and thus strange to the mind of most people so that it appears to be secure, it is still a weak password if a proactive password checker includes that word in its dictionary file. On the contrast, some really weak passwords with low entropy could be considered to be "good" by proactive password checkers. This is a common shortcoming within current proactive password checking. Wu [13] also observed this.

In our experiment, all existed password checkers including *ProCheck* failed to catch weak passwords like `a198b53`, which are of low entropy. Ironically, `12345abc` could be rejected as a weak password by some checkers, but `12a3b4c5` would be accepted as a good one by all checkers. Similarly, some checkers could easily reject `12345ab`, but failed to catch `12a34b5`. This kind of failure comes from the rational: 1) the common practice for password management is always based on dictionary attack to search weak passwords, and those passwords that cannot be cracked by dictionary-based attacks are usually considered to be secure; and consequently, 2) current proactive checkers mainly (if not totally) rely on dictionary-based checking, and most low-entropy passwords are ignored.

These low-entropy passwords constitute a new class of weak passwords, and need to be properly addressed.

3 ENTROPY BASED PROACTIVE PASSWORD CHECKING: AN EXAMPLE

We propose to use entropy based proactive password checking to detect the abovementioned new class of weak passwords, and allow only high entropy passwords. Moreover, we propose to dig out effective patterns of weak passwords with low entropy as the first step of performing entropy-based proactive checking.

Until now, a few password patterns have been exploited to recognize weak passwords by current password checkers. For example,

- Minimum password length;
- All digits or all punctuation characters;
- Calendar dates or phone numbers;
- Adjacent keys, such as `12345ab`, `12345abc`, `ehm12345`, `abcdefgh`.

Nonetheless, those used patterns are of a very limited number and type, and they cannot tackle many other weak passwords with low entropy. On the other hand, the simple password policy described in the beginning of Section 2 leads to a too strong pattern to be acceptable. In this section, we take 7-character alphanumeric password as an example of seeking for weak patterns of low entropy passwords. We deliberately choose 7-character case-insensitive alphanumeric password as our example, because they are widely used in real life, though many systems like Unix and Windows NT support case sensitive passwords. Empirical data showed that users generally avoided using the shift key, and 86% passwords cracked in Wu's experiment could be typed without it [13]. This might partially explain that. On the other hand, the password scheme of Novell Netware is case insensitive, so there are more passwords that fall into this category in a Netware environment.

3.1 Different Distribution Areas for 7-Character Passwords

We denote the permutation operation by P(). Consider a 7-character alphanumeric password. It may reside in one of the eight exclusive **distribution areas** defined as follows.

1). P(7a): all 7 characters are alphabetic
2). P(6a+1n): 6 alphabetic and 1 numeric
3). P(5a+2n): 5 alphabetic and 2 numeric
4). P(4a+3n): 4 alphabetic and 3 numeric
5). P(3a+4n): 3 alphabetic and 4 numeric
6). P(2a+5n): 2 alphabetic and 5 numeric
7). P(1a+6n): 1 alphabetic and 6 numeric
8). P(7n): all 7 characters are numeric

Table 1 shows the search space and cost of each of these eight areas. The search cost is benchmarked with an attacking speed of 4.7μs per try, which is measured for the Novell Netware password hash algorithm on a Pentium 333 Windows NT machine.

Among these eight areas, it takes the highest percentage of the full search (36^7) to cover the P(5a+2n) area, which is an area with the highest entropy for a 7-character alphanumeric password, or the most secure area in terms of brute force attacks. Similarly, P(6a+1n) is the second strongest area. However, either P(7n), P(1a+6n) or P(2a+5n) is a relatively **weak area** where passwords are with low entropy. As shown in Table 1, there is a clear division between high and low entropy areas. We use a dashed line to mark the division.

3.2 Different Pattern Distributions for 7-Character Passwords

In this section, we look into the distribution of different password patterns in the P(7a), P(6a+1n), P(5a+2n) and

P(4a+3n) areas. Table 2 ~ 5 list each possible password pattern in each area, along with its search cost as a percentage of this area and of the whole search space of 36^7. In each of these tables, there is a clear division that separates strong and weak password patterns in that area. We also use a dashed line to show the division boundary in each table.

Table 1. Different distribution areas for 7-character passwords

Areas	Search space	= Value	Percentage	Cracking Time		
				minutes	hours	days
Full	36^7	78,364,164,096	100.00%	6,138.53	102.31	4.26
P(7a)	26^7	8,031,810,176	10.25%	629.16	10.49	0.44
P(6a+1n)	(C_7^1 *10) * 26^6	21,624,104,320	27.59%	1,693.89	28.23	1.18
P(5a+2n)	(C_7^2 *10^2)*26^5	24,950,889,600	31.84%	1,954.49	32.57	1.36
P(4a+3n)	(C_7^3 *10^3)*26^4	15,994,160,000	20.41%	1,252.88	20.88	0.87
P(3a+4n)	(C_7^4 *10^4)*26^3	6,151,600,000	7.85%	481.88	8.03	0.33
P(2a+5n)	(C_7^5 *10^5)*26^2	1,419,600,000	1.81%	111.20	1.85	0.08
P(1a+6n)	(C_7^6 *10^6)*26	182,000,000	0.23%	14.26	0.24	0.01
P(7n)	10^7	10,000,000	0.01%	0.78	0.01	0.00
Total:		78,364,164,096	100.00%	6,138.53	102.31	4.26
Speed(s/try):	4.70E-06					

Table 2. Pattern Distributions in the P(7a) Area

Patterns	Search space	= Value	Percentage (Value/P(7a))	Cost (Value/36^7)
All for P(7a)	**26^7**	**8,031,810,176**	**100.00%**	**10.25%**
1. No repeated character	P_{26}^{7}	3,315,312,000	41.28%	4.23%
2. Only one repeated character				
One occurs twice	$C_{7}^{2} * P_{26}^{6}$	3,481,077,600	43.34%	4.44%
One occurs three times	$C_{7}^{3} * P_{26}^{5}$	276,276,000	3.44%	0.35%
One occurs four times	$C_{7}^{4} * P_{26}^{4}$	12,558,000	0.16%	0.02%
One occurs five times	$C_{7}^{5} * P_{26}^{3}$	327,600	0.00%	0.00%
One occurs six times	$C_{7}^{6} * P_{26}^{2}$	4,550	0.00%	0.00%
One occurs seven times	$C_{7}^{7} * 26$	26	0.00%	0.00%
3. Two repeated characters				
each occurs twice	$C_{26}^{5} *10*(7! / (2!)^2)$	828,828,000	10.32%	1.06%
one twice, another three times	$C_{7}^{2} * C_{5}^{3} * P_{26}^{4}$	75,348,000	0.94%	0.10%
one twice, another four times	$C_{7}^{2} * C_{5}^{4} * P_{26}^{3}$	1,638,000	0.02%	0.00%
one twice, another five times	$C_{7}^{2} * P_{26}^{2}$	13,650	0.00%	0.00%
each occurs three times	$C_{26}^{3} *3*(7! / (3!)^2)$	1,092,000	0.01%	0.00%
one three, another four times	$C_{7}^{3} * P_{26}^{2}$	22,750	0.00%	0.00%
4. Three repeated characters				
each occurs twice	$C_{26}^{4} *4*(7! / (2!)^3)$	37,674,000	0.47%	0.05%
two occurs twice, another three times	$C_{26}^{3} *3*(7! / (2!2!3!))$	1,638,000	0.02%	0.00%
	Total:	**8,031,810,176**	**100.00%**	**10.25%**

Table 3. Pattern Distributions in the P(6a+1n) Area

Patterns	Search space	= Value	Percentage (Value/P(6a+1n))	Cost (Value/36^7)
All for P(6a+1n)	**(C_7^1 *10) * 26^6**	**21,624,104,320**	**100.00%**	**27.59%**
1. No repeated alphabetic	(C_7^1 *10)* P_{26}^6	11,603,592,000	53.66%	14.80%
2. Only one repeated alphabetic				
one occurs twice	(C_7^1 *10)*(C_6^2 * P_{26}^5)	8,288,280,000	38.33%	10.57%
one occurs three times	(C_7^1 *10)*(C_6^3 * P_{26}^4)	502,320,000	2.32%	0.64%
one occurs four times	(C_7^1 *10)*(C_6^4 * P_{26}^3)	16,380,000	0.08%	0.02%
one occurs five times	(C_7^1 *10)*(C_6^5 * P_{26}^2)	273,000	0.00%	0.00%
one occurs six times	(C_7^1 *10)*26	1,820	0.00%	0.00%
3. Two repeated alphabetic				
each occurs twice	(C_7^1 *10)*(C_{26}^4 * C_4^2 * 6! / (2! * 2!))	1,130,220,000	5.23%	1.44%
one twice, another three times	(C_7^1 *10)* (C_6^2 * C_4^3 * P_{26}^3)	65,520,000	0.30%	0.08%
one twice, another four times	(C_7^1 *10)* (C_6^2 * P_{26}^2)	682,500	0.00%	0.00%
Each occurs three times	(C_7^1 *10)* (C_{26}^2 * 6! / (3! * 3!))	455,000	0.00%	0.00%
4. Three repeated alphabetic				
Each occurs twice	(C_7^1 *10)* (C_{26}^3 * 6! / (2! * 2! * 2!))	16,380,000	0.08%	0.02%
	Total:	**21,624,104,320**	**100.00%**	**27.59%**

Table 4. Pattern Distributions in the P(5a+2n) Area

Patterns	Search space	= Value	Percentage (Value/P(5a+2n))	Cost (Value/36^7)
All for P(5a+2n)	**(C_7^2 *10^2)*26^5**	**24,950,889,600**	**100.00%**	**31.84%**
1. No repeated alphabetic	(C_7^2 *10^2)* P_{26}^5	16,576,560,000	66.44%	21.15%
2. Only one repeated alphabetic				
one occurs twice	(C_7^2 *10^2)*(C_5^2 * P_{26}^4)	7,534,800,000	30.20%	9.62%
one occurs three times	(C_7^2 *10^2)*(C_5^3 * P_{26}^3)	327,600,000	1.31%	0.42%
one occurs four times	(C_7^2 *10^2)*(C_5^4 * P_{26}^2)	6,825,000	0.03%	0.01%
one occurs five times	(C_7^2 *10^2)*26	54,600	0.00%	0.00%
3. Two repeated alphabetic				
each occurs twice	(C_7^2 *10^2)*(C_{26}^3 * 3 * 5! / (2! * 2!))	491,400,000	1.97%	0.63%
one twice, another three times	(C_7^2 *10^2)*(C_5^2 * P_{26}^2)	13,650,000	0.05%	0.02%
	Total:	**24,950,889,600**	**100.00%**	**31.84%**

Table 5. Pattern Distributions in the P(4a+3n) Area

Patterns	Search space	=Value	Percentage (Value/P(4a+3n))	Cost (Value/36^7)
All for P(4a+3n)	**(C_7^3 *10^3)*26^4**	**15,994,160,000**	**100%**	**20.41%**
1. No repeated alphabetic	(C_7^3 *10^3)* P_{26}^4	12,558,000,000	78.52%	16.03%
2. One repeated alphabetic				
one occurs twice	(C_7^3 *10^3)*(C_4^2 * P_{26}^3)	3,276,000,000	20.48%	4.18%
one occurs three times	(C_7^3 *10^3)*(C_4^3 * P_{26}^2)	91,000,000	0.57%	0.12%
one occurs four times	(C_7^3 *10^3)*26	910,000	0.01%	0.00%
3. Two repeated alphabetic				
each occurs twice	(C_7^3 *10^3)*(C_{26}^2 *4! / (2! * 2!))	68,250,000	0.43%	0.09%
	Total:	**15,994,160,000**	**100.00%**	**20.41%**

3.3 A Simple Checking Algorithm

The above analysis shows that there are clear divisions both between strong and weak password areas, and between strong and weak password patterns. It is obvious that a password that falls into the following two categories must be with relatively low entropy:

- Passwords in the P(7n), P(1a+6n), or P(2a+5n) area, i.e., passwords that have 5 or more numerical characters;
- Passwords that are in the P(7a), P(6a+1n), P(5a+2n) or P(4a+3n) area and have two or more repeated alphabetic, or one alphabetic occurring three or more times.

Traditionally, brute force attack ignored these divisions and aimlessly searched each candidate in the full password space. That is the reason that it couldn't efficiently crack weak passwords like `ca12612`. If the search order of a brute force attack is weighted by entropy, and low-entropy parts are a first priority, then low-entropy passwords are likely cracked by this smart brute force attack at a cost far less than expected.

To prevent all low-entropy passwords defined above, a simple but efficient algorithm can be defined as follows.

```
PROCEDURE: Proactive_checking_for_7pwd ()
INPUT: char * password
Begin procedure
  Scan each character in password;
  IF (there are ≥ 5 numeric characters)
  THEN reject;
  ELSE /* matching two legal patterns */
       IF (there is ≤ 1 repeated alphabetic)  AND
          (occurrence of the repeated alphabetic ≤ 2)
       THEN accept;
       ELSE reject;
       ENDIF
  ENDIF
End procedure
```

This is not an alternative algorithm, but a complementary one to improve dictionary-based checking for 7-character alphanumerical passwords. Some password checkers like *Npassword* [9] could reject passwords with three or more adjacent repeated characters, which, however, only constitute a small subset of weak passwords covered by our algorithm.

If passwords generated from all identified weak patterns are included into a dictionary, entropy based checking can be achieved by the traditional dictionary-based approach. Nonetheless, pattern-based entropy checking has the following obvious advantages: 1) it can not only save the storage, but also improve the checking speed by reducing the dictionary search space; and typically, 2) its algorithm is efficient due to its simplicity.

4 SUMMARY AND SUGGESTIONS FOR FURTHER WORK

Although new password techniques have emerged, proactive password checking is still a desirable method to improve password security in real life. Unfortunately, current checking algorithms mainly (if not totally) depend on dictionary-based checking, and they often fail to filter some weak passwords with low entropy. We suggested the use of entropy-based proactive password checking to address this new class of weak passwords. To dig out effective patterns of weak passwords with low entropy was proposed as the first step of performing entropy-based proactive checking, and an example was given.

Entropy-based checking is not an alternative method, but a complementary one to improve dictionary-based checking. Good proactive password checking = dictionary-based checking + entropy-based checking. What we have done here is only a first step towards a full search of weak password patterns for entropy based checking. It is easy to extend our analysis and algorithm for, say, passwords with eight or more characters, or passwords that consist of only alphabetic and punctuation characters, and it is useful to develop entropy-based password checking algorithms for Unix and Windows NT passwords to improve their current dictionary-based algorithms. Moreover, our discussion of classes of passwords is ad hoc. It would be interesting to look for a generic means.

5 ACKNOWLEDGMENTS

The author thanks Alasdair Grant for providing benchmarking data for the Novell Netware password hash algorithm. The discussion with Wenbo Mao helped the author to clarify some points. The comments from anonymous reviewers and participants of NSPW'01 improved this paper. Mpiti Lenkoe helped correct some grammar errors in a previous version of this paper.

6 REFERENCES

[1] Steven M. Bellovin and Michael Merritt, Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks, IEEE Symposium on Research in Security and Privacy, May 1992. pp.72-84.

[2] F Bergadano et al. High dictionary compression for proactive password checking, ACM trans. on info and system security Vol.1, No.1, Nov. 1998

[3] F Bergadano et al. Proactive password checking with decision trees, 1997 ACM conference on computer and communications security, 1997, Zurich

[4] Burton Bloom. Space/time trade-offs in hash coding with allowable errors, CACM, 13(7): 422-426, July 1979

[5] C. Davies and R. Ganesan. BApasswd: A new proactive password checker. In 16th National Computer Security Conference, pages 1--15, Baltimore, MD, Sept. 1993

[6] DV Klein. Foiling the Cracker; A Survey of, and Improvements to Unix Password Security, Proceedings of the USENIX Security Workshop. Portland, Oregon: USENIX Association, Summer 1990; expanded as a technical report from SEI, 1992

[7] Alec Muffett. Crack 4.0, 5.0, almost everywhere in the internet

[8] Alec Muffett. CrackLib: a proactive password sanity ibrary. http://www.users.dircon.co.uk/~crypto/download/cracklib,2.7.txt

[9] Npassword source code (Latest version: npasswd-2.X.tar.gz). at http://www.utexas.edu/cc/unix/software/npasswd/dist/npasswd-2.05.tar.gz, 2000

[10] S. Patel, Number theoretic attacks on secure password schemes. IEEE Symposium on Security and Privacy, 1997

[11] E. H. Spafford. OPUS: Preventing Weak Password Choices, Computers and Security 11(3), pp. 273-278, 1992

[12] T. Wu, The Secure Remote Password Protocol, in Proceedings of the 1998 Internet Society Symposium on Network and Distributed System Security, San Diego, CA, Mar 1998, pp. 97-111.

[13] T. Wu, A Real-World Analysis of Kerberos Password Security, Proceedings of the 1999 Network and Distributed System Security Symposium, February 3-5, 1999

[14] Jianxin (Jeff) Yan, Alan Blackwell, Ross Anderson and Alasdair Grant. The Memorability and Security of Passwords -- Some Empirical Results. Technical Report No. 500, Computer Laboratory, University of Cambridge, 2000. http://www.ftp.cl.cam.ac.uk/ftp/users/rja14/tr500.pdf