

# POSH: A GENERALIZED CAPTCHA WITH SECURITY APPLICATIONS

by

WASEEM S. DAHER

Bachelor of Science, Massachusetts Institute of Technology (2007)

Submitted to the Department of Electrical Engineering and Computer Science  
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2008

Copyright 2008 Waseem S. Daher.

The author hereby grants to M.I.T. permission to reproduce and distribute publicly  
paper and electronic copies of this thesis document in whole or in part.

This work is licensed under the Creative Commons Attribution License. To view a  
copy of this license, visit <http://creativecommons.org/licenses/by/3.0/> or send a  
letter to Creative Commons, 559 Nathan Abbott Way, Stanford, CA 94305.

Author .....  
Department of Electrical Engineering and Computer Science  
May 23, 2008

Certified by .....  
Ran Canetti  
Visiting Scientist  
Thesis Supervisor

Certified by .....  
Ronald L. Rivest  
Viterbi Professor of Computer Science  
Thesis Supervisor

Accepted by .....  
Arthur C. Smith  
Chairman, Department Committee on Graduate Theses



# POSH: A GENERALIZED CAPTCHA WITH SECURITY APPLICATIONS

by

WASEEM S. DAHER

Submitted to the Department of Electrical Engineering and Computer Science  
on May 23, 2008, in partial fulfillment of the  
requirements for the degree of  
MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

## Abstract

A **puzzle only solvable by humans**, or **POSH**, is a prompt or question with three important properties: it can be generated by a computer, it can be answered consistently by a human, and a human answer cannot be efficiently predicted by a computer. In fact, a POSH does not necessarily have to be verifiable by a computer at all. One application of POSHes is a scheme proposed by Canetti et al. that limits off-line dictionary attacks against password-protected local storage, without the use of any secure hardware or secret storage.

We explore the area of POSHes, implement several candidate POSHes and have users solve them, to evaluate their effectiveness. Given these data, we then implement the above scheme as an extension to the Mozilla Firefox web browser, where it is used to protect user certificates and saved passwords. In the course of doing so, we also define certain aspects of the threat model for our implementation (and the scheme) more precisely.

Thesis Supervisor: Ran Canetti  
Title: Visiting Scientist

Thesis Supervisor: Ronald L. Rivest  
Title: Viterbi Professor of Computer Science



## Acknowledgments

This work would not have been possible without Ran Canetti, who provided guidance, constructive feedback, and all-around enthusiasm, literally every step of the way. For that, I owe him a great deal. Additionally, without Ron Rivest’s involvement, this project would never have gotten off the ground. For many students, “Canetti” and “Rivest” are names one only sees on conference papers, in RFCs, or in books – I consider myself lucky to have had the opportunity to work with them.

I would also like to thank the members of the M.I.T. Student Information Processing Board for providing support, entertainment, and occasionally even good advice. In particular, I would like to thank Jessica McKellar for drawing a large number of the faces used in the Face Recognition puzzle, and Jeff Arnold for constantly reminding me exactly how much (or how little) time I had before each of the thesis deadlines.

I would be remiss if I did not thank all of the volunteers who contributed data to this project by solving puzzles. Hopefully some of them found it fun, though I expect a larger number did it solely as a favor to me.

Listing everyone who has helped me over the course of my academic career would be impossible. I am indebted to many people for their encouragement and support, and this is especially the case for my family. Perhaps someday we’ll all sit down together with this document, and I’ll finally explain exactly what I’ve been up to for this past year.



# Contents

<b>1</b>	<b>Introduction</b>	<b>11</b>
<b>2</b>	<b>Techniques in Password Security</b>	<b>15</b>
2.1	Early work and current practice . . . . .	15
2.2	A Web-based solution . . . . .	17
2.3	Password-based key exchange . . . . .	19
2.4	Inkblots for stronger passwords . . . . .	21
2.5	Other graphical password schemes . . . . .	22
<b>3</b>	<b>CAPTCHA Overview</b>	<b>25</b>
3.1	Fuzzy Text Recognition . . . . .	25
3.2	Other Approaches . . . . .	26
3.3	Corpus-based CAPTCHAs . . . . .	27
<b>4</b>	<b>Evaluating POSHes</b>	<b>29</b>
4.1	Word Association . . . . .	29
4.2	Word Grouping . . . . .	32
4.3	Draw-a-Noun and Draw-a-Phrase . . . . .	33
4.4	Inkblot . . . . .	36
4.5	Face Recognition . . . . .	38
4.6	CAPTCHA . . . . .	39
<b>5</b>	<b>Mitigating Offline Dictionary Attacks</b>	<b>41</b>
5.1	Description of CHS proposal . . . . .	41

5.2	POSH requirements imposed by CHS . . . . .	42
5.3	Using POSH-derived keys for encryption . . . . .	43
<b>6</b>	<b>An implementation for Firefox</b>	<b>45</b>
<b>7</b>	<b>Conclusions and Future Work</b>	<b>49</b>

# List of Figures

2-1	Pinkas and Sander’s scheme for online dictionary attacks . . . . .	18
2-2	A naïve solution to the EKE problem . . . . .	19
2-3	The Encrypted Key Exchange protocol . . . . .	20
2-4	Three “Inkblots” designed to encourage stronger passwords . . . . .	21
2-5	A Déjà Vu password prompt . . . . .	23
2-6	The Face and Story password prompts . . . . .	23
3-1	An ESP-Pix CAPTCHA . . . . .	26
4-1	Word Association POSH example . . . . .	30
4-2	Word Association answer distribution . . . . .	31
4-3	Word Grouping POSH example . . . . .	32
4-4	Word Grouping answer distribution . . . . .	33
4-5	Draw-a-Noun and Draw-a-Phrase POSH examples . . . . .	34
4-6	Inkblot POSH example . . . . .	36
4-7	Inkblot answer distribution . . . . .	37
4-8	Face Recognition POSH example . . . . .	38
4-9	Traditional CAPTCHA POSH example . . . . .	39
5-1	The CHS scheme . . . . .	42
6-1	Firefox dialog modification . . . . .	46



# Chapter 1

## Introduction

In this paper, we investigate what we call a **puzzle only solvable by humans** or **POSH**. Like CAPTCHAs, POSHes are “puzzles” that can be generated by computers and answered by humans, such that the answer cannot be efficiently predicted by a computer [vABL04]. However, unlike a CAPTCHA, a POSH need not have a computer-verifiable answer; the only requirement is that a person’s answer to a given puzzle is consistent over time.

Everything that is a CAPTCHA is by definition also a POSH, but not vice versa. For example, “What is your favorite food?” is not a valid CAPTCHA (there is no “correct” answer) but is a valid POSH. We are interested in exploring this space further: what features make for a desirable POSH? What constraints affect the POSHes we can reasonably create? Which POSHes are actually fun to solve?

The study of POSHes is not just one done for the sake of academic curiosity: they have interesting security applications. One such application is a scheme that mitigates dictionary attacks on password-protected local storage, in the absence of secure hardware or secret storage. Additionally, the Inkblot scheme of [SS04] is essentially an application of POSHes to the problem of generating stronger passwords. Finally, “security questions” used to verify the user’s identity (commonly seen on bank websites) also fit our definition of a POSH, to name a few examples.

# A password security application for POSHes

It appears to be a fact of life that user-chosen passwords are weak. They tend to be low-entropy, and tend to be drawn from a small dictionary. Assigning strong, random passwords to users often provokes resentment or, worse, cheating – they just write them down on notepads left near their computers, largely defeating the point. There must be a better alternative to weak passwords: it is clear that the human mind can be incredibly inventive and can perform complex computations, but these powers simply don't seem to be used for passwords. Is there a better interface for harnessing this as-of-yet untapped potential? We contend that using POSHes provides the beginnings of an answer, and is a step in the right direction.

We target the following threat model: the user has a laptop whose hard drive is encrypted with a secret key derived from the user's password. The attacker steals the laptop, mounts the hard drive in his own machine, and launches an *offline* dictionary attack against the secret key. More generally, this is the problem of mitigating dictionary attacks against password-protected local storage, without secure hardware or secret storage.

Protecting against dictionary attacks in this restricted setting seems impossible. However, in 2006, Canetti, Halevi, and Steiner proposed a scheme that makes use of POSHes to accomplish these goals [CHS06]. The gist of the scheme is that testing each password requires the solutions to a unique set of POSHes, even for the offline attacker. Since solving POSHes require humans (by definition!), this dramatically slows down the rate at which an attacker can make guesses, crippling the attack. However, the system was never implemented, and if one is truly concerned with practicality, implementation is everything.

## This paper

A POSH has very simple requirements: a human must be able to answer it consistently, and a computer must not be able to efficiently predict human answers. With

an eye towards practical application, we consider the following metrics on which a POSH should be evaluated.

**Consistency** When presented with the same POSH, how reproducible is a user's answer? The level of consistency will clearly vary across POSHes, and the acceptable level will vary by application (some may be more lenient than others).

**Entropy** By entropy, we mean two things: First, do different people answer the same POSH in the same way? Second, how hard is it for an adversary to guess the user's answer, given the POSH? For the POSH to be good in this respect, there should be a sizable set of possible answers, and furthermore it should be computationally hard to predict the user's answer from the POSH alone. These questions become particularly relevant in security or cryptographic applications, where we want to use POSHes to generate secrets.

**Fun** A good POSH should not require any specialized knowledge or complicated actions, and should have a low barrier to entry. Relatedly, it should not be too time-consuming or tedious. It is worth noting that fun may also markedly affect the other parameters. For example, a fun puzzle may be easier to remember.

**Ease of generation** How difficult is it to generate a given POSH? Can it be generated given only randomness, or does it require a precomputed/pregenerated corpus? If so, how much space does such a corpus occupy? What is the yield of this corpus – is an entry in the corpus required for each new POSH, or can several POSHes be generated from a given entry? Can the POSH be generated or stored without its answer? ([CHS06] requires this property)

**Implementation** Finally, how easy is it to implement? Does it require complex and elaborate graphics, or can it be implemented for a text-only system? How accessible is it?

With these requirements in mind, we explore seven different POSHes:

1. **Word Association:** The user is presented with six words selected at random, and is asked for a word he/she associates with each of them.
2. **Word Grouping:** The user is presented with six words and is asked to section them off into two different groups, categorized any way they wish.
3. **Draw-a-Noun:** The user is presented with a noun, and asked to draw a 10x10 black-and-white icon representing it.
4. **Draw-a-Phrase:** Like Draw-a-Noun, only the prompt takes the form of a noun, followed by a preposition, followed by another noun, and a 15x15 grid is given.
5. **Inkblot:** Inspired in part by [SS04], the user is presented with a randomly generated black-and-white figure reminiscent of a Rorschach inkblot, and asked for a word or phrase that he/she associates with it.
6. **Face Recognition:** The user is presented with a cartoon face with randomly selected features, and asked for an association – based on empirical evidence that humans are much better than machines at recognizing faces [SBOR06].
7. **CAPTCHA:** A POSH in the spirit of a traditional CAPTCHA – the user is presented with fuzzy letters and asked to type them in.

We implemented each of these POSHes and placed them on `puzzles.mit.edu`, a Web site we created to test their effectiveness. We then took the Word Association POSH and wrote an extension for the Mozilla Firefox web browser that uses the CHS scheme to protect the user’s saved passwords and certificates.

Chapter 2 provides an overview of existing password security techniques, and Chapter 3 provides an overview of existing CAPTCHAs. Chapter 4 discusses the implementation and results of our seven POSHes. Chapter 5 presents the CHS scheme for mitigating dictionary attacks in more detail and discusses the additional requirements it places on POSHes, and Chapter 6 and describes the implementation of the scheme for the Mozilla Firefox Web browser. Finally, Chapter 7 concludes, providing suggestions for future work to be done in this area.

# Chapter 2

## Techniques in Password Security

There are  $(26 + 26 + 32)^8 = 84^8 \approx 2^{51}$  possible eight-character passwords that can be made using uppercase letters, lowercase letters, and punctuation. In the ideal world, people would choose passwords uniformly from this set. However, passwords tend to be low-entropy, and are typically taken from a relatively small domain (a *dictionary*) with high probability. This is problematic because it lends itself to a *dictionary attack* – one where the attacker typically succeeds after trying all the passwords in the dictionary (as opposed to trying all  $2^{51}$  passwords).

Dictionary attacks come in two varieties: *online* and *offline*. In an online dictionary attack, each password attempt is sent to a verifier (a program not under the attacker’s control, a remote machine, etc.) to check. In an offline attack, the attacker knows something that allows him to determine the password’s correctness by himself (for example, we might know how to compute  $h$ , and that  $h(p) = x$ ). In this chapter, we survey a variety of techniques designed to address these problems.

### 2.1 Early work and current practice

In the 1960s, it was common practice for login usernames and passwords to be stored unencrypted in a *password file* designed to be unreadable to the system’s users. When a user attempts to login with password  $p'$ , the system looks up the password  $p$  associated with the username. If  $p = p'$ , the login is successful, otherwise the user is

rejected.

This practice was changed when a notable bug on the CTSS time-sharing system caused the “Message of the Day,” shown at login, to be overwritten with a copy of the password file, exposing everyone’s password [Cor91].

It became clear that another approach – one in which the password file need not remain secret – was necessary. Evans et. al provide such an approach in [EKW74]. In the password file, store  $user, key$  where  $key = h(p)$  and  $h$  is a one-way, collision-resistant hash function. When a user attempts to log in with password  $p'$ , compute  $h(p')$  and succeed if  $h(p') = key$ . Publishing the password file does not lead to catastrophic results, since one cannot log in by submitting  $key$ , and computing its inverse, the actual password, is hard.

Unfortunately, this method is susceptible to an offline dictionary attack, if the attacker can get ahold of the password file. The idea here is that the attacker pre-computes  $h(p')$  for all the passwords in his dictionary. Once the attacker gets the password file, he merely has to scan his list of hashes to see if there are any matches. This attack is quite feasible, even on a PDP-11 [MT79].

However, precomputation can be made intractable the use of a *salt*. The idea behind this technique – used by Unix – is that, when a user account is created, a fresh random salt  $s$  is selected. The password file then looks like  $user, salt, key$ , where  $key = h(p|salt)$  and  $|$  denotes concatenation. If the salt is 32 bits long and the attacker wants to precompute the password file entry for  $p'$ , he must calculate  $2^{32}$  passwords:  $h(p'|0)$  up to  $h(p'|2^{32} - 1)$ . In this case, generating such a dictionary before having seen the password file becomes prohibitively expensive.

If the attacker has the password and is targeting a specific user, though, he does not need to compute  $2^{32}$  hashes for every password attempt, because the salt is known – it is not encrypted. So the attacker may still hash his entire dictionary with the given salt to try to crack the password, but this has to be done per user.

However, even this is not strong enough. One way to further mitigate such attacks is to make the hashing step extremely slow and expensive. For example, making  $key = \text{KDF}(p, salt)$ , where KDF is a complicated *key derivation function* [Kal00]. A

candidate for such a function might be something that iteratively applies SHA-256 on  $p|salt$  thousands of times. Evaluating KDF on a given input requires a significant amount of computation, and thus it is hoped that doing it for every password in the attacker’s dictionary will take far too long.

Such a scheme, in the vein of the “pricing via processing” solution to junk mail of [DN93], is common today. However, it is somewhat distasteful – too few iterations make computation trivial for a powerful adversary, and too many makes logging on slow for an authorized users. Furthermore, it is essentially useless in the stolen laptop scenario, since the attacker has all the time in the world (and can parallelize). Finally, this also results in something of an arms race: as computers get faster, more iterations become necessary.

Can we do better? In limited contexts, the answer appears to be yes.

## 2.2 A Web-based solution

Pinkas and Sander examine the problem of defending against online dictionary attacks in [PS02], specifically focusing on the problem of defending a Web site that requires login.

They begin by describing two commonly-implemented countermeasures against online dictionary attacks, and refuting them: delayed response, and account locking. The strategy of delayed response is one in which the server simply waits for a second or two before responding “yes” or “no” to the login request – this means that an attacker can only try passwords at a system-specified, slow, rate. The account locking strategy is one in which a given account is locked out for, say, five hours after three incorrect password attempts.

These countermeasures do well if the attacker is trying to compromise a specific account, but fail if the goal is to compromise *any* account. For example, the attacker can, in parallel, request  $(u_1, p_1)$ ,  $(u_1, p_2)$ , etc., which circumvents the delayed response scheme. Furthermore, if the list of usernames is large (as it is for most web services), there are enough that one can pause for five hours between each attempt, since there

are plenty of other names to try. In addition, account locking provides a vector for a denial-of-service attack (and a customer service nightmare).

However, this problem is solvable, by using a test that is easy for computers to generate and verify, but hard for them to solve – in other words, a CAPTCHA (more on this in Chapter 3). A simple solution is the following: prompt the user for a CAPTCHA solution. If the CAPTCHA is solved correctly, prompt for their username and password, otherwise give up on the process.

This turns out to be somewhat arduous for a serious production-quality Web site, though, because generating CAPTCHAs for every single login attempt may be too expensive for the Web server, and because it also represents added work that the user most likely would be reluctant to deal with every time. Instead, Pinkas and Sander propose another scheme, which we’ve simplified and described here, in Figure 2.2:

1. Prompt for the username and password
2. Check to see if the client has a valid cookie
3. If he does:
  - (a) Password is correct: **Accept**
  - (b) Password is incorrect: show the user a CAPTCHA and **Reject** regardless of its result
4. If he does not:
  - (a) Password is correct: show the user a CAPTCHA. If it is solved correctly, **Accept** and set a cookie, else **Reject**
  - (b) Password is incorrect: show the user a CAPTCHA and **Reject** regardless of its result

Figure 2-1: Pinkas and Sander’s scheme for online dictionary attacks

This approach is desirable because legitimate users of the system only have to solve CAPTCHAs in two cases: when they log in from a new computer, or when their cookie expires (after, say, 100 successful logins). The chief solvers of the CAPTCHAs are people who mistype their passwords and the would-be attackers. Pinkas and Sander also present a modification to the system where incorrect password attempts only

solve CAPTCHAs *some* of the time, without significantly weakening these desirable properties, the details of which are fleshed out in [PS02].

## 2.3 Password-based key exchange

A related problem is the following: how can two parties (e.g. a client and server) agree on a strong session key given only a weak shared secret, like a password? What's more, we'd like the protocol to protect the password from offline dictionary attacks. Bellare and Merritt discuss this in [BM92, BM93], the result of which is a protocol for *encrypted key exchange*, or EKE.

We present an abbreviated version of the notation Bellare and Merritt use in describing EKE, in Table 2.1.

Table 2.1: Bellare and Merritt's notation for EKE

$A, B$	Users of the system (Alice and Bob)
$P$	The password – a shared secret
$K$	A random secret key (for a symmetric cryptosystem)
$R_A, R_B$	Random exponents
$K[info]$	Symmetric-key encryption of “info” using key $K$
$K^{-1}[info]$	Symmetric-key decryption of “info” using key $K$
$c_A, c_B$	Random challenges generated by $A$ and $B$ , respectively
$\alpha, \beta$	Base and modulus for discrete exponentiation

A naïve approach to such a protocol is illustrated in Figure 2-2.

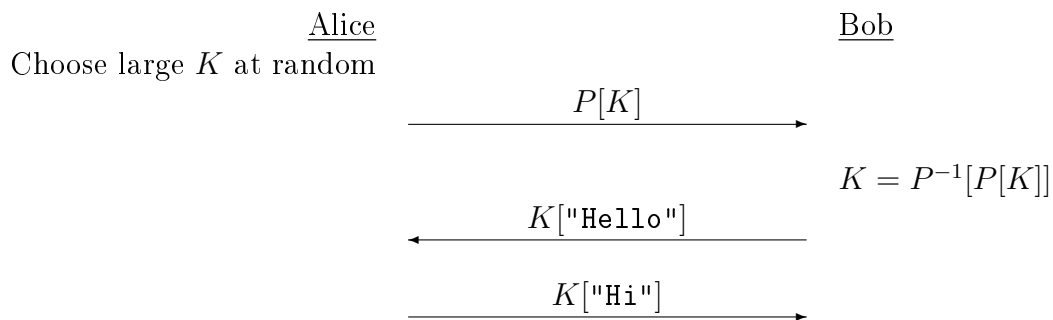


Figure 2-2: A naïve solution to the EKE problem

This solution, however, is susceptible to an offline dictionary attack against  $P$ . The attacker guesses a  $P'$  and computes  $K' = P'^{-1}[P[K]]$ . With  $K'$ , the attacker decrypts Bob's first response. If it is a valid protocol message, then  $P'$  is the real password. But all is not lost: a solution to the problem exists, and involves the use of Diffie-Hellman key exchange [DH76]. It is presented in Figure 2-3.

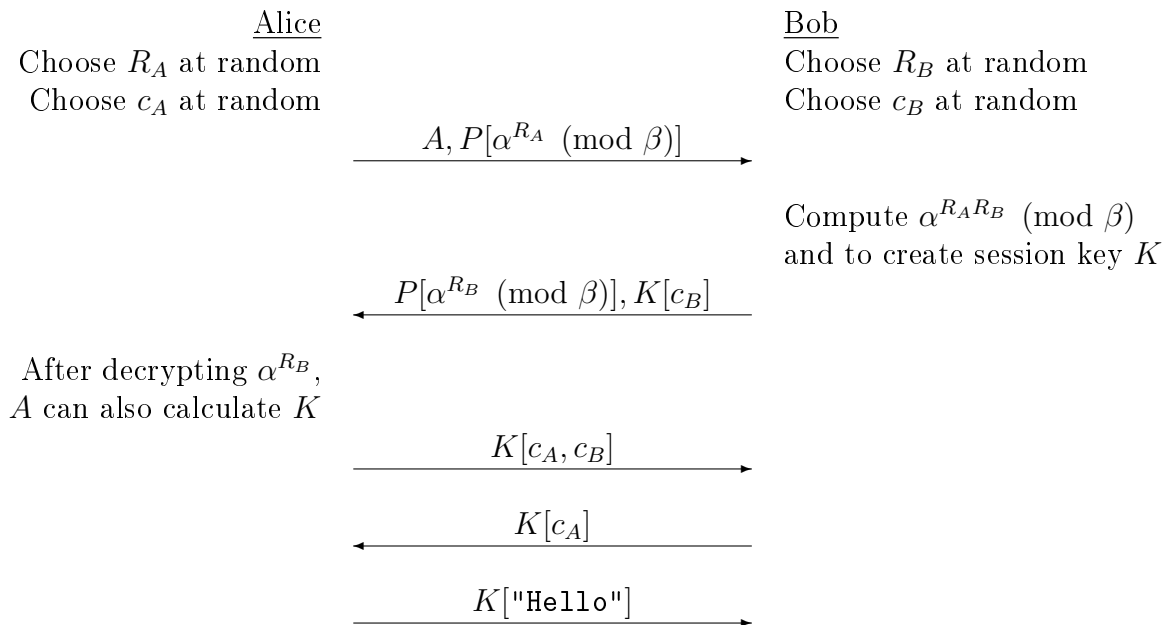


Figure 2-3: The Encrypted Key Exchange protocol

There are a few things to notice. The first, and most interesting, is that this approach has fixed the offline dictionary attack problem. A password guess  $P'$  gives the attacker two candidate values:  $\alpha^{R_A'}$  and  $\alpha^{R_B'}$ . However, these are uninteresting: since these values are essentially chosen at random (since  $R_A$  and  $R_B$  are chosen randomly), they do not leak any real information about the correctness or incorrectness of the guess. What's more, an attacker cannot use them to compute a candidate  $K'$  (to see if a valid protocol message is being sent after the key exchange completes), under the assumption that the discrete log problem is hard.

It is also worth noting that the third and fourth messages (with challenges and responses) provide protection against a man-in-the-middle attack, and that a modification to EKE exists that permits the use of a hashed version of the password rather

than the password itself [BM93].

Significant further work has been done in this area, as well. Many people, including [BPR00], have provided provide a set of formal definitions and analysis for the EKE protocol and related solutions. In particular, [CHK<sup>+</sup>05] provides definitions and a way to solve this problem that is secure under the universally composable security framework [Can01], which means that the protocol remains secure even when run concurrently with arbitrary other protocols.

## 2.4 Inkblots for stronger passwords

Another approach of note is one which tries to improve the *quality* of the user-selected password. The system, Inkblots [SS04], generates a series of images that look like Rorschach Inkblots, and displays them to the user. The user is supposed to come up with an association for each inkblot, and, to create a password, takes the first and last letters from each association and concatenates them. So, if inkblots 1, 2, and 3 reminded the user of “helicopter”, “hippo doing stretches”, and “crab”, the resulting password would be “hrhscb”. These inkblots can be seen in Figure 2-4.



Figure 2-4: Three “Inkblots” designed to encourage stronger passwords

Here, unlike any other strategy shown above, the user is essentially given an algorithm for picking a password based on computer input, rather than coming up with his or her own, and having it strengthened by other mechanisms. While innovative, it is unclear to what extent rigorous analysis can be performed on the Inkblots – if the majority of them look similar, it is not clear that they will be a good source of entropy. However, preliminary studies done (and referenced) in [SS04] look promising.

That point notwithstanding, the Inkblots display an approach different from any discussed above; one in which the user is encouraged to create a password more resilient to dictionary attacks, rather than simply making dictionary attacks more difficult.

## 2.5 Other graphical password schemes

The history of graphical password schemes is a long and storied one. Unfortunately, few of these schemes seem to have caught on in practice, but we review a few that seem especially topical below.

Jermyn et al. attempt to combat weak passwords by proposing Draw-a-Secret [JMM<sup>+</sup>99], a graphical scheme in which the user’s drawing is converted into a bit-string that serves as the user’s password. They implemented the system as a login mechanism for a PDA, and the system is further analyzed in [TvO04]. It is quite similar to our Draw-a-Noun POSH. One important difference, however, is that we tell the user what to draw (e.g. “draw a house”), whereas Draw-a-Secret does not.

Also in the space of graphical passwords, Dhamija and Perrig propose Déjà Vu [DP00]. The system works as follows. At account setup time, the user is presented with  $n$  images, and selects  $k$  of them to be his password. Later, when he wishes to log on, the system presents  $n$  images (possibly the same, possibly new ones, as long as the  $k$  are included) to the user, and he must select the same  $k$  to log on successfully. In a sense, this is also POSH – the prompt is a set of  $n$  images, and the question is “Which of these look familiar?” An example is shown in Figure 2-5.

Finally, Davis et al. perform a study [DMR04] of two graphical password schemes, Face and Story. In Face, like Déjà Vu, at initialization time, the user is shown  $n$  faces and selects  $k$  of them for her password. To log in, the user is shown  $n$  faces and has to select his  $k$ . In Story, the user is shown  $n$  images and has to select  $k$  of them *in order*, that are supposed to tell a story. Examples of these two are shown in Figure 2-6.

In Story, for example, one might pick the man (in the top-center), the keys (in the bottom-right), and the car (in the center), because the “story” is “A man gets keys to

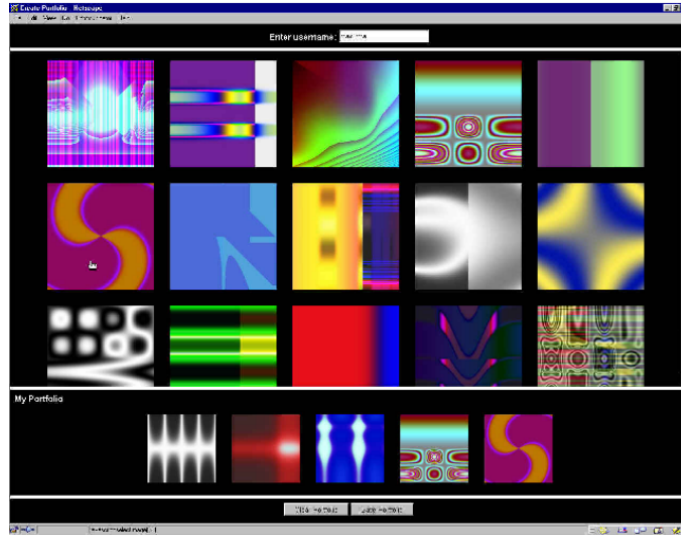


Figure 2-5: A D  j   Vu password prompt

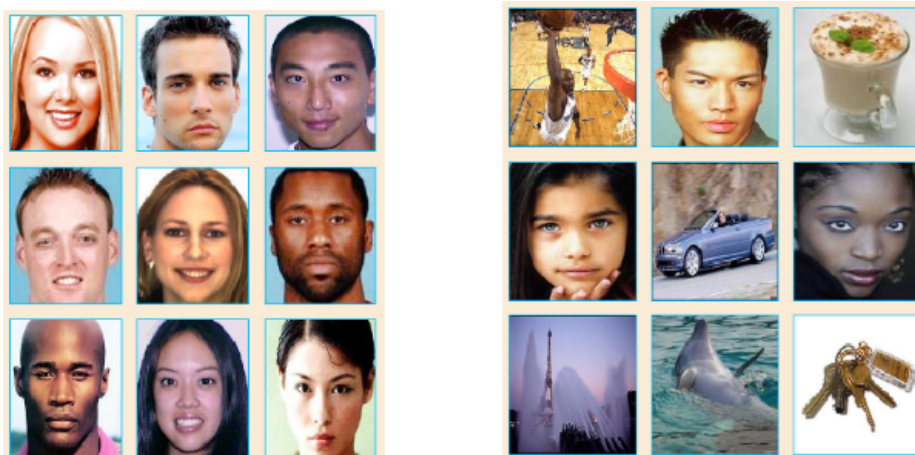


Figure 2-6: The Face and Story password prompts

go drive his car.”

Their results (particularly applicable to the Face Recognition POSH) are the following. People tend to prefer attractive faces, often from their own ethnicities, so the effective entropy of the scheme is *significantly* reduced for Face. For Story, few users actually construct a story – they can easily remember the image, but do not often recall the image order correctly.



# Chapter 3

## CAPTCHA Overview

The idea of a test administered by a *computer*, that can distinguish computers from humans, appears to have first been proposed in [Nao96]. There, Naor proposes a handful of candidate puzzles, all believed to be easy for humans and hard for computers, and mostly from the area of computer vision. Potential tests that are suggested are: gender recognition, facial expression recognition, detecting nudity, reading handwriting, recognizing speech, filling in the blanks, et cetera. However, the paper is largely theoretical; there is little to no mention of implementation details.

Such tests received many names: Turing tests, reverse Turing tests, and CAPTCHAs, but one thing that emerged relatively quickly was that a test based on the difficulty of recognizing characters was both straightforward to implement and enjoyed a fairly high success rate.

### 3.1 Fuzzy Text Recognition

The first successful implementation of this idea was a system where the user is presented with fuzzy letters, and asked to read them. Andrei Broder et. al [LABB01] are believed to be the first to have actually implemented such a system, while at AltaVista. There, it was used to foil bots trying to submit URLs into the AltaVista index.

This work was soon followed by others, including [CFB01]. There, Coates et. al

suggest the idea of “Pessimial Print” – text manipulated in ways designed to be hard for OCR software. In [vABL04], Luis von Ahn coins the term CAPTCHA, and proposes a similar scheme, named “Gimpy.”

After it became clear that CAPTCHAs were useful and not just a research curiosity, they began to spring up everywhere, largely in a security context: as a way to prevent denial-of-service attacks or comment spam, or as a way of slowing down brute-force attacks.

### 3.2 Other Approaches

In the same work, von Ahn et. al propose some a handful of other CAPTCHAs: “Bongo,” a visual pattern recognition puzzle, “Eco,” a sound recognition puzzle, and “Pix,” a label-the-object puzzle. In “Pix,” the user is presented with blurred images of an object, and asks “What are these images of?” An implementation of Pix, known as ESP-Pix, is shown in Figure 3-1.



Figure 3-1: An ESP-Pix CAPTCHA

Speech CAPTCHAs, where words are said over white noise, seem to show promise [KLS02], but getting sound to work just right in a Web browser, e.g., can be an inconvenience at best and a major annoyance at worst.

Despite the existence of these other techniques, fuzzy text CAPTCHAs remain the most popular. While simple to implement, these (and any graphical scheme) are inaccessible to the blind. The existence of a robust text-based CAPTCHA would certainly solve this problem, though it is also still believed to be quite challenging, despite some promising work in the area [God02]. For more, [LS05] provides a nice survey of CAPTCHA techniques and implementations.

### 3.3 Corpus-based CAPTCHAs

Another interesting trend is the move towards corpus-based CAPTCHAs, like reCAPTCHA [vA07]. reCAPTCHA is a project to digitize books by turning words that trigger OCR errors into CAPTCHAs. The scheme works as follows: the user is presented with two blurry words, one that the computer knows and one that it does not, in a random order. The human reads both words, and submits the answer. If his answer to the known word is correct, we use his answer to the unknown word as a vote for the actual correct answer.

Interestingly enough, this is more of a POSH than a CAPTCHA, since the correct answer is *not* known to the computer. However, this problem is solved by the sheer number of users of the system – if enough people agree on an answer, the system decides that it is correct.

The downside, of course, is that these types of CAPTCHAs can only be generated and verified by someone with access to a specific (often large) corpus. So, to use reCAPTCHA on one’s site, one must trust von Ahn et. al to return the correct result – a requirement that does not exist with traditional CAPTCHA.



# Chapter 4

## Evaluating POSHes

Recall that our goal was to test the effectiveness of seven POSHes: Word Association, Word Grouping, Draw-a-Noun, Draw-a-Phrase, Inkblot, and a traditional CAPTCHA. We set up a Web site, `puzzles.mit.edu`, where we implemented the POSHes and invited users to solve them. Each user is required to create an account before solving POSHes, but no other data is required (the user has the option of telling us his/her e-mail address).

After logging in, the user is presented with a list of all the POSHes that he or she can solve, and can elect to solve brand-new ones, or re-solve previously-seen ones (in order to test recall for a given POSH). In addition, the user may skip a POSH if he/she does not want to solve it. After solving a POSH, the user is shown everyone else's answers to the just-solved puzzle.

As of this writing, the Web site has been available for slightly over two months, has approximately 80 users, and has seen roughly 2500 page loads. We now describe each POSH in more detail, along with its results.

### 4.1 Word Association

Six words are selected at random from a corpus, and the user is asked for a word that he or she associates with each of them. We chose a corpus of 505 words derived from a Pictionary wordlist [Eng07], with the rationale that they would be reasonably

common and have strong associations. Figure 4-1 shows one set of these POSHes being solved.

diameter	circle
rocket	ship
megaphone	announcement
pound	dollar
morse code	SOS
wheelbarrow	hay
<input type="submit" value="Submit"/>	

Figure 4-1: Six solved Word Association POSHes

Word Association was the most popular POSH we studied: 56 users collectively solved 1,163 puzzles (in 193 submissions, since each submission contains six puzzles), suggesting that it has a low barrier to entry and is fairly fun.

**Consistency** The consistency of answers was also fairly good: 19 users re-solved 331 puzzles, and 64% of the new responses were the same as the user’s original response, suggesting that Word Association is *reasonably* memorable without any concerted effort or practice. (The numbers are unfavorably skewed by users that entered some word *a* on a first attempt, and then a different word *b* for all subsequent attempts – a somewhat common failure mode.)

**Entropy** The data suggest that some words have very strong associations, but most do not. We examine the 28 most popular of these POSHes (the ones answered by the most unique users) to get a better sense of this. For each of the POSHes, we found the most popular answer, and then computed the percentage of overall answers that the most common answer represents. These results are shown in Figure 4-2. For example, “four eyes” appears on the very far right, since nearly everyone associated “glasses” with it. On the other hand, no two users came up with the same association for “wine glass”: answers ranged from “classy” and “merlot” to “harmonica”<sup>1</sup>.

---

<sup>1</sup>If someone can figure out the train of thought that leads to this answer, please let us know.

Pinning down the entropy in such a puzzle is difficult: a naïve attacker would potentially have to try a large subset of all English words. For reference, Shakespeare’s complete works contain approximately 29,000 words [Sha], and the language has grown rapidly since then. A more clever attacker might dramatically reduce this number by guessing common associations more intelligently (synonyms, antonyms, words that frequently appear by the prompt, etc.). Analyzing this case well is more challenging.

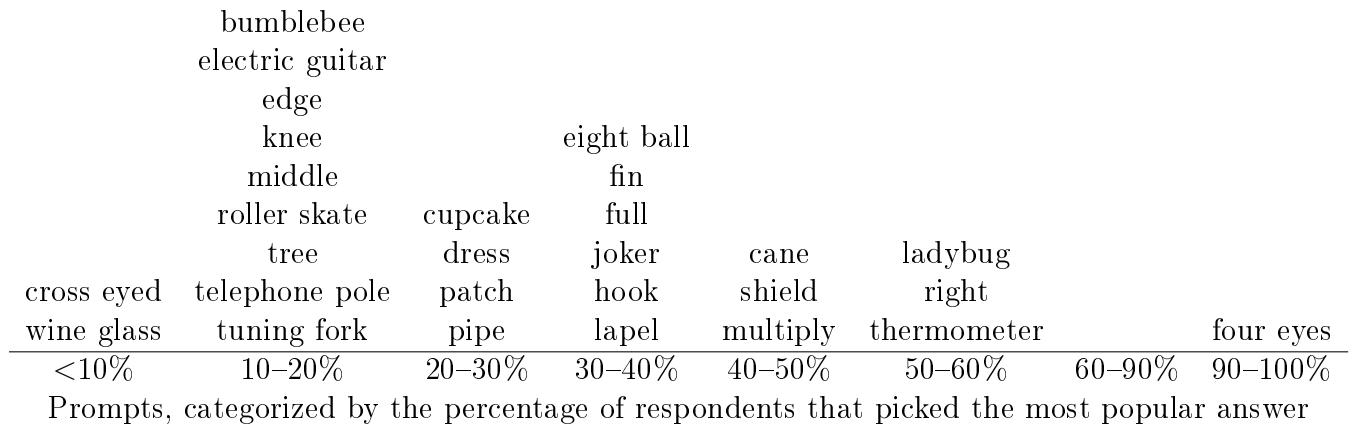


Figure 4-2: Word Association answer distribution

**Ease of generation** Word Association does less well on ease of generation. We can only generate 505 unique puzzles – the size of the corpus. Corpora this small lend themselves to attempts at “exhaustive enumeration,” by which we mean humans building a large database of reasonable answers, which a computer could use to “solve” the POSH. Furthermore, making the corpus much bigger runs the risk of prompting users with words that they may not know (and thus, prompts for which they do not have memorable associations).

**Implementation** The implementation is straightforward, and has several desirable properties: the task is easy (and something users don’t find foreign), and the user interface is simple and accessible – which means that it can work on screen readers or in a text-only environment like a login prompt. Showing each puzzle is also simple

and space-efficient; one merely chooses a word at random out of a list.

## 4.2 Word Grouping

Using the same corpus as above, the user is presented with six words, and is asked to divide the group into two subsets, using any categorizing the user wishes. Figure 4-3 shows an example. Word Grouping was unfortunately not that popular – 28 unique users solved 98 puzzles (in 49 submissions).

butterfly	<input checked="" type="radio"/> Set 1	<input type="radio"/> Set 2
dog	<input checked="" type="radio"/> Set 1	<input type="radio"/> Set 2
jar	<input type="radio"/> Set 1	<input checked="" type="radio"/> Set 2
lens	<input type="radio"/> Set 1	<input checked="" type="radio"/> Set 2
spider web	<input checked="" type="radio"/> Set 1	<input type="radio"/> Set 2
trunk	<input type="radio"/> Set 1	<input checked="" type="radio"/> Set 2

Figure 4-3: A solved Word Grouping POSH, grouped by natural vs. man-made

**Consistency** Though not especially fun, Word Grouping seems somewhat memorable without much practice: of the 10 users that went back and re-solved 34 puzzles, 58% of them were recalled perfectly. We suspect that this rate can be boosted with a tiny bit of practice, since users merely have to remember *how* they reached their categorizations (e.g. alive or inanimate), not the actual results, and can apply the same technique to many POSHes.

**Entropy** In principle, each POSH has  $2^6$  possible outputs, and we expect to see a large amount of variation. In practice, this seems to hold up as well – few of the 8 most popular puzzles had duplicates in their solutions. A breakdown is shown in Figure 4-4.

**Ease of generation** Word Grouping has some of the same limitations on its corpus – it cannot become too large or users will not recognize some of the words. However,

Prompts, categorized by % users selecting mode	the percentage of respondents that picked the most popular answer Prompts
00–10%	bigfoot, cloverleaf, odd, point, shoulder, starfish broken heart, drip, fold, hat, icicle, screwdriver horseshoe, knee pad, right hand, skull, tuning fork, two left feet
10–20%	direction, drawbridge, label, small, stereo, twins
20–30%	concave, dart, eat, pelican, pencil, vertical eye, hook, oval, rocket, slot machine, spider web
30–40%	beard, book, elephant, headband, necktie, northstar
40–50%	
50–60%	hot air balloon, screwdriver, see, stoplight, sunglasses, tic tac toe
60–70%	
70–80%	
80–90%	
90–100%	

Figure 4-4: Word Grouping answer distribution

the problematic feature of Word Association was that each word’s answer was independent of all of the other words; not so with Word Grouping. To solve the puzzles, one theoretically needs to use information from all of the words. As a result, a corpus of size  $n$  and a puzzle of size  $k$  yields  $\binom{n}{k}$  possible puzzles, or, in our case,  $\binom{505}{6}$ , which is a very desirable result.

**Implementation** The implementation of Word Grouping has all of the same advantages of Word Association: it only requires a text-based interface, and the task is simple. Constructing the puzzles from the corpus is also simple – just select six at random.

### 4.3 Draw-a-Noun and Draw-a-Phrase

Since the Draw-a-Noun and Draw-a-Phrase POSHes are so similar, we combine their discussion into one section. In Draw-a-Noun, one word is selected at random from our Pictionary corpus, and the user is asked to illustrate it, using a 10×10 black-and-white grid. Draw-a-Phrase works very similarly, only two words are selected from the corpus, and joined with one preposition, selected from a list of 50. The user is given

this word/preposition/word prompt, and asked to illustrate it with a  $15 \times 15$  grid. Examples of both of these are shown in Figure 4-5.

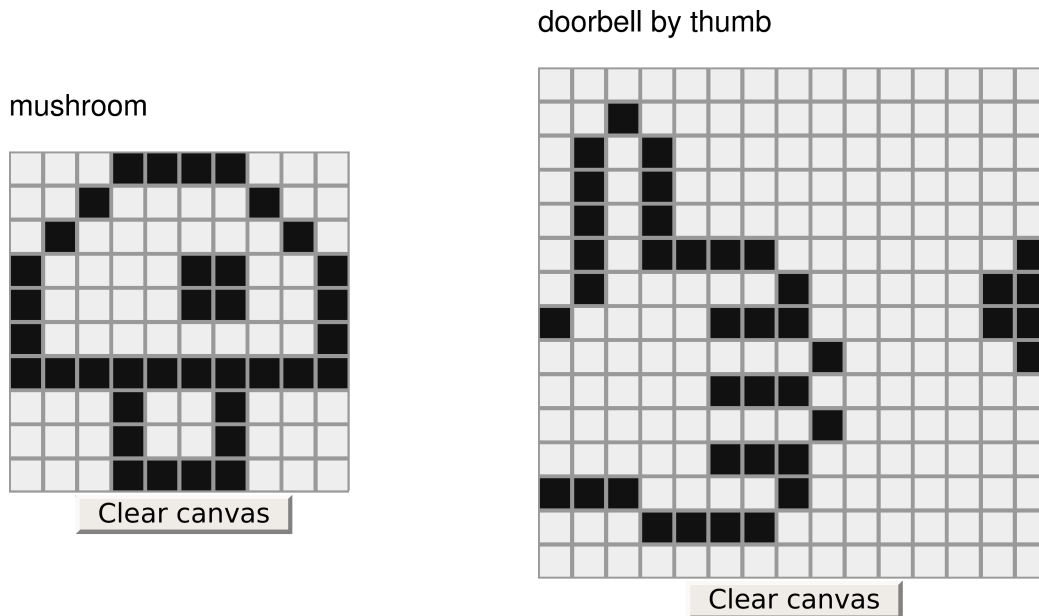


Figure 4-5: A solved Draw-a-Noun (left) and Draw-a-Phrase (right) POSH

Draw-a-Noun was definitely among the most popular of the POSHes: 30 users collectively drew 258 drawings (in 129 submissions), suggesting that even though illustrating is more work than some of the other POSHes, it is entertaining. However, this amusement is not without limit: Draw-a-Phrase was by far the least popular POSH we studied: 14 users collectively drew 60 drawings (in 30 submissions), which is quite low compared with the other puzzles. It is fair to conclude that this POSH was simply not fun.

**Consistency** While fun, unfortunately, Draw-a-Noun does not appear to be especially memorable. Seven users went back and re-drew 26 puzzles. Of these, only slightly over 15% of all re-draws were identical to the first inputs. One problematic aspect of the results is that, occasionally, the image was the same, but shifted by a pixel. Fortunately, addressing the shifted-image problem is quite doable, using techniques similar to those in [JMM<sup>+</sup>99]. The gist of the idea is to derive the POSH solution from the start and end points of every line drawn, and to do so at a more coarse resolution than that of the canvas. For example, we could represent our  $10 \times 10$

grid as a  $5 \times 5$  grid for the purposes of tracking lines. If that is done, we can tolerate an off-by-one-pixel error in three directions, for any given point. The downside to such an approach, however, is a decrease in the amount of available entropy.

As for Draw-a-Phrase, few users bothered to go back and re-draw puzzles in the first place – the five that did went back to draw 16 images. Of those, no one was able to successfully recreate a drawing, suggesting that recalling a  $15 \times 15$  image given these prompts is a difficult task.

**Entropy** For Draw-a-Noun, there are, in principle, 100 bits to be set with each illustration, so there are  $2^{100}$  possible drawings. Realistically, not all of these will occur, so we offer the following approximation: Suppose a drawing contains eight lines. Each of these lines starts and ends at an arbitrary coordinate in the canvas, of which there are 100. So we have somewhere near  $10 \times 100 \times 100 = 100,000$  possible answers.

Additionally, we expect that the chance of two people illustrating the same thing the same way should be quite small. Of the 30 unique prompts, only two had some identical illustrations: three illustrators for “full” drew an all-black icon, and the two illustrators of “negative” drew a five-pixel-long minus sign in the middle of the canvas. No two people illustrated the exact same thing for any other prompt. This trend is reflected in the Draw-a-Phrase results as well: no two users had identical drawings.

**Ease of generation** Generation of Draw-a-Noun works exactly like Word Association, and suffers from the same flaws – the number of unique puzzles is equal to the size of the corpus, and the corpus must additionally remain fairly small or the prompts will become too hard to draw.

Draw-a-Phrase, on the other hand, gets a better yield from the corpus. Since a puzzle is constructed with noun/preposition/noun, the number of possible puzzles is  $|N| \cdot |P| \cdot |N|$ , or, in this case,  $505 \times 50 \times 505 = 12,751,250$  puzzles.

**Implementation** Draw-a-Noun and Draw-a-Phrase are departures from our previous puzzles in that they are somewhat graphical. They require some way of marking

grid squares either off or on, so it could still be done in a text-based environment like a terminal. However, they are not very accessible and would not interact well with a screen reader, for example.

## 4.4 Inkblot

In a change of pace from word prompts, we decided to explore some more graphical options. In Inkblot, the user is presented with a randomly generated image reminiscent of a Rorschach inkblot (using a modified version of [Hal01]), and asked for a word or phrase that he/she associates with it. Figure 4-6 shows one such inkblot. Inkblots were fairly popular – 32 unique users solved 200 puzzles (in 100 submissions), so it seems safe to conclude that they are fairly fun.



Figure 4-6: An Inkblot POSH

**Consistency** Ten of the users went back and re-solved 60 of the inkblots, and roughly 64% of the new solutions matched the originals perfectly. Of those that did not, about half were “one-bit” errors (e.g. making a word plural rather than singular), and the others were just different associations. While not perfect, this POSH shows relatively promising rates of recall. (User studies in [SS04] also confirm this observation).

**Entropy** Like Word Association, some inkblots had very strong associations for many people, but most do not. Of the 20 unique puzzles solved by our users, the large majority (14) had no common answers, and in the remaining ones, the mode answer only represented approximately 10% of the total answers, with three exceptions: inkblots that looked a lot like a face, crab, and heart. Figure 4-7 shows the breakdown in more detail.

In terms of space of possible answers, we expect the user inputs to provide roughly as much entropy as a randomly chosen English phrase, while remaining resistant to the frequency-based attacks that pose problems for Word Association.

Prompts, categorized by % users selecting mode	Prompts
00–10%	10011, 14605, 15531, 18248, 27590, 35195, 35849, 49118, 53619, 55076, 57475, 61492, 62647, 62844
10–20%	26535, 46998, 56230
20–30%	49032 (“face”)
30–40%	
40–50%	
50–60%	38145 (“crab”)
60–70%	
70–80%	
80–90%	
90–100%	9027 (“heart”)

Figure 4-7: Inkblot answer distribution

**Ease of generation** Inkblots do extremely well on ease of generation. They require only one argument to generate: a random seed. As a consequence, no corpus needs to be stored at all; each inkblot can be generated on-the-fly, fairly efficiently. In addition, one can generate an extremely large number of puzzles, limited primarily by the random number generator.

**Implementation** One possible improvement to the scheme would be the use of colors in the inkblots, as in [SS04], in the hopes of making them more memorable. One of the weaknesses of the Inkblot scheme is its reliance on graphics – it cannot

be straightforwardly adapted to work in a terminal or text-only device, and it has no hope of working with screen readers.

## 4.5 Face Recognition

Continuing the trend of graphical POSHes, the next one we explored was Face Recognition. The motivation here is that humans are much better at recognizing faces than computers are, and that perhaps we can use that advantage to make a good POSH. The user is presented with a cartoon face with randomly selected features (hair, ears, eyes, face, mouth, and nose all selected independently), and prompted with “This face makes me think/feel...”. An example is shown in Figure 4-8.

Face Recognition was only moderately popular – 30 users solved 130 puzzles (in 65 submissions), which puts it in fifth place (above Word Grouping and Draw-a-Phrase).

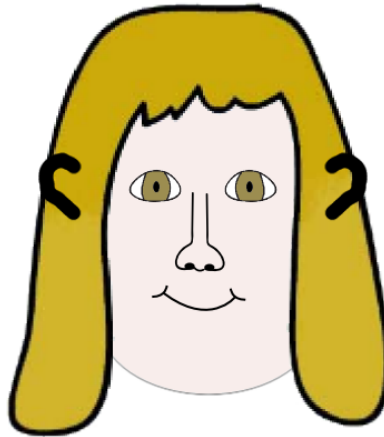


Figure 4-8: A Face Recognition POSH

**Consistency** Perhaps an unfavorable indication of their level of fun, not many users went back and re-solved these puzzles. For the five that re-examined 20 puzzles, only 35% of the answers matched the originals perfectly. Those that didn't were generally fairly off-target, unlike the Inkblots.

**Entropy** Most faces tended to be high-entropy. Of the 27 unique faces seen by the users, all but one yielded different responses. The one that wasn't was thought to be

“french” by two users, but this represented less than 10% of the replies for the face. (Other replies included “mime” and “kiss”, to name a few).

We expect the answers to be a limited subset of English: descriptions of thoughts and feelings. The literature also suggests that the entropy is most likely not evenly distributed by feature – eyebrows seem to matter most for face recognition [SBOR06].

**Ease of generation** The features we varied were ears (2 versions), eyes (5), facial shape (7), hair (6), mouth (7), and nose (5), allowing for  $2 \times 5 \times 7 \times 6 \times 7 \times 5 = 14,700$  unique faces. This POSH has the nice property that adding additional features has very high yield. Mixing and matching features is a much more efficient construction than having to provide a corpus of completely-drawn faces.

**Implementation** An implementation suggestion that might make the faces more memorable would be the use of photographs of human features, rather than cartoons. Another variation on this scheme could pose the question “Who or what does this face remind you of?” in which case we could add names to our list of possible answers. Additionally, like Inkblot, Face Recognition’s reliance on graphics hinders its accessibility.

## 4.6 CAPTCHA

The final POSH we examine is one that is a traditional CAPTCHA – that is to say, fuzzy letters that the user is asked to read. We use a slightly modified variant of [Nie08] to generate the CAPTCHAs, and an example is shown in Figure 4-9.



Figure 4-9: A “traditional CAPTCHA” POSH

CAPTCHA did fairly well – 35 users solved 190 CAPTCHAs (in 95 submissions), suggesting the not-terribly-surprising result that they are straightforward to solve.

**Consistency** Ten users went back and re-solved 46 CAPTCHAs, and 100% of them solved them the exact same way, which does not come as a huge surprise.

**Entropy** Nearly everyone provided the same solutions for all of the CAPTCHAs; there was essentially no variation. In two of the 22 unique puzzles solved by the most users, one user confused a “z” with an “x” and an “o” with an “a,” but all of the other answers were the same.

While there was basically no inter-user variation, CAPTCHA does quite well in terms of the amount of entropy it is capable of providing. There are 44 possible letters in each position (upper and lowercase, with commonly-confused pairs like C/G, I/l, Q/O, h/b removed), and five possible positions, yielding  $44^5 \approx 2^{27}$  possible puzzle answers.

**Ease of generation** CAPTCHAs are, by design, fairly easy to generate – they simply require the text to be rendered and some randomness. At first blush, this POSH passes our tests with flying colors: it does not require a corpus, and it is easy to make many of them. However, there is a bit of a catch: generating CAPTCHAs also requires their answers. If we need them without answers, as we do in [CHS06], we need to do a bit more. This is described in more detail in Chapter 5.

**Implementation** Like its fellow graphical POSHes, this style of CAPTCHA cannot be used in a text-only environment, and thus is not especially accessible.

# Chapter 5

## Mitigating Offline Dictionary Attacks

### 5.1 Description of CHS proposal

Recall that our goal is to mitigate offline dictionary attacks, without any sort of secret storage, secure hardware (like a smart card or hardware token), or assistance by a remote server (unlike in the scheme of [PS02]). Canetti, Halevi, and Steiner present such a scheme in [CHS06], which we informally describe here.

Let us assume that the system has the ability to present a large number of possible POSHes – either by having computed them in advance or by generating them on-the-fly. Furthermore, suppose that all the POSHes are stored in a large array, and are addressable by an index.

The user begins the login process by supplying a username and a password. We compute  $p_1, p_2, \dots, p_n = \text{Expand}(pw)$ , where `Expand` is a deterministic function that takes the password as an argument, and returns the indices of several POSHes that the user needs to solve. `Expand` should behave like a random function, in the sense that each different password should map to a new, random set of indices. More specifically, [CHS06] shows that `Expand` needs to behave like a good expander graph.

The user solves the puzzles, and computes  $k = \text{Extract}(pw, s_1, s_2, \dots, s_n)$ , where  $s_i$  is the solution to POSH  $p_i$ . With appropriate selection of `Expand` and `Extract`,  $k$  should appear pseudorandom. In particular, it should be hard to guess  $k$ . We then use  $k$  to encrypt our local storage (or as the actual “password”, if we use the scheme

solely for authentication). Figure 5-1 illustrates the process.

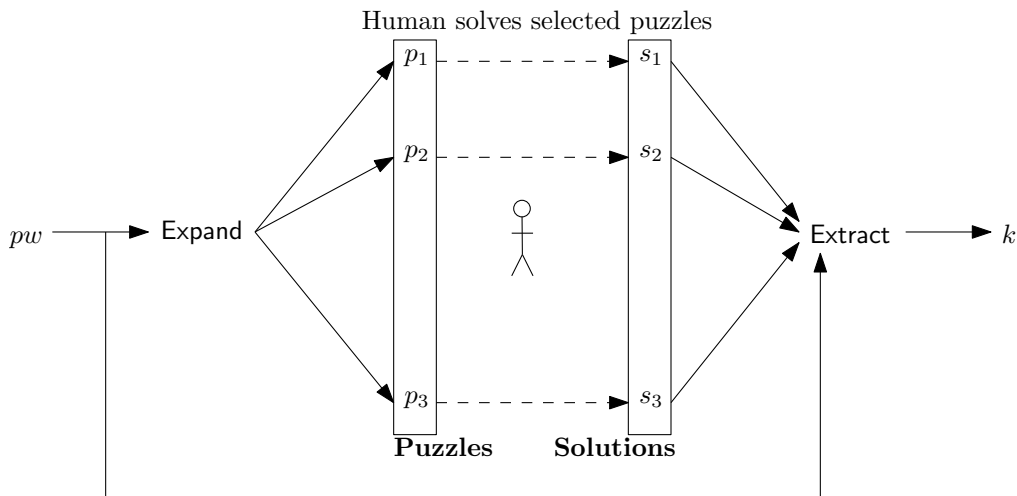


Figure 5-1: The CHS scheme

If we do this, an attacker mounting an offline dictionary attack against your data has two options: one is to try to guess the secret key  $k$  directly. Given that  $k$  is hard to guess, this will be challenging – comparable to a brute-force search of the keyspace. The other alternative is to make guesses at the (probably) weak password  $pw$  – but each guess of  $pw$  also requires a human to solve new POSHes, significantly slowing down the attack.

## 5.2 POSH requirements imposed by CHS

Recall that POSHes need not have a computer-verifiable answer. In some cases, however, they do – the process of generating a CAPTCHA involves knowing its solution. In the CHS scheme, the answer to a given POSH *must not* be known at login-time, because if it is, it can be accessed by the attacker. In other words, the POSH generator cannot simply take  $p_i$  and display a CAPTCHA of  $p_i$ , because the attacker also knows  $p_i$  given a password guess (and thus knows the CAPTCHA’s solution). Contrast this with the Inkblot POSH. For the Inkblot POSH, it is perfectly acceptable to seed the Inkblot generator with  $p_i$ , because knowing  $p_i$  does not help the attacker determine the answer.

This is not a fatal flaw in CAPTCHAs, however. The solution is to pre-generate and store a large number of CAPTCHAs, discard their answers, and shuffle them around. Now, selecting the CAPTCHA with index  $p_i$  tells the attacker no information about its solution. The downside is the additional space requirement required to store all of the CAPTCHA images.

We also require an additional property of our POSHes: that they not be *malleable* – that is, a human solution to POSH  $a$  should not provide a computer any help in predicting the solution to POSH  $b$ . If this is not the case, the offline attacker can defeat the scheme by having humans solve a small subset of the POSHes, and using those solutions to predict reasonable answers for the others.

### 5.3 Using POSH-derived keys for encryption

Traditional encryption schemes assume that the randomness present in the secret key is uniform. The output of the `Extract` function may be random, but we may not have guarantees about the uniformity of this randomness over its many bits. As a consequence, we need to use a scheme that provides security even in the presence of this potentially weak key. An example of such a scheme is [CD08], but further research in this direction is warranted.



# Chapter 6

## An implementation for Firefox

The Mozilla Firefox Web browser can remember usernames and passwords for users, as well as private keys and personal certificates. It stores these in its Software Security Device. All of the information in the Software Security Device is encrypted by the user’s Master Password [Fou01].

This setup is exactly one in which the CHS scheme is appropriate: a malicious user or program can copy the encrypted contents of Software Security Device from the user’s disk for later offline attacks. Firefox already has a mechanism for prompting the user for his/her Master Password, so implementing the scheme is simply a matter of writing an extension to replace the correct dialog boxes. For comparison, the original “Get Master Password” dialog box and our replacement are shown side-by-side in Figure 6-1. Similar modifications were required for the “Set Master Password” and “Remove Master Password” dialog boxes.

We chose the Word Association POSH for use in our implementation, because it was popular and simple, both in UI and in actual implementation. For `Expand`, we computed the SHA-1 hash of the password, divided the resultant 160 bits into six 26-bit numbers, took those numbers modulo the size of our corpus, and used them as indices into our word list. `Extract` was also very straightforward – we concatenated the password and all of the POSH answers together, and passed the result on to Firefox, as though it were entered into the original dialog box.

Note that, in our dialog box, the POSHes are displayed at the same time as



Figure 6-1: Firefox “Get Master Password” window before and after our changes

the password is. As the password is entered, the POSHes change (the SHA-1 is recomputed with every keystroke). This has the (unexpected!) benefit of giving an authorized user faster feedback about an incorrect password. When the correct password is entered, the usual POSHes appear; when it isn't, foreign ones do, and the fact that an error has been made is quickly apparent.

The implementation itself was reasonably straightforward. In general, windows in Firefox are specified using XUL – an XML-based user interface language. Anything specified by XUL can be overridden by an extension; thus, one dialog can be seamlessly replaced with another.

Unfortunately, as of this writing, the specifics of Firefox's internals mean that the “Get Master Password” dialog box does not use XUL (it is created directly in `nsNSSCallbacks.cpp`), meaning that it cannot be easily replaced. To address this,

we submitted a patch [Dah08] that makes Firefox use a XUL window instead (and thus allows extensions to replace it). It has yet to be accepted into the mainline branch, though we expect the patch to be non-controversial, since the user experience does not change at all (only our *extension* modifies the dialog box), and installed extensions must be fully trusted by the user, so the change does not pose a security risk. In the interim, we developed the extension against a patched local build of Firefox.

The “Change” and “Remove” windows did use XUL, and thus did not require recompilation to modify. Once the patch was applied, three new XUL files needed to be created (`getmp.xul`, `setmp.xul`, and `changemp.xul`) to serve as the replacement dialog boxes. These specify how the window should look; the actual logic needs to be specified elsewhere, in JavaScript. To accomplish this, we wrote `password.js`, which performs the logic for all three windows (which seems reasonable since there is a good deal of overlap: changing and removing the password essentially first require you to get it). It is here that the actual implementation of the CHS scheme is provided. A breakdown of all of these files, and their associated sizes, is provided in Table 6.1.

Filename	Size (lines)	Purpose
Patch	-51,+44	Call a user-specified XUL window to get the password, instead of the hard-coded prompt
<code>getmp.xul</code>	+74	XUL Specification for Get Master Password window
<code>changemp.xul</code>	+139	XUL Specification for Change Master Password window
<code>removemp.xul</code>	+87	XUL Specification for Remove Master Password window
<code>password.js</code>	+239	Code to make windows functional and perform CHS
<code>sha1.js</code>	+100	A SHA-1 implementation (used in <code>password</code> )
<code>words.js</code>	+505	Word Association corpus (used in <code>password</code> )

Table 6.1: Firefox modifications

The Firefox extension is licensed under the GPL and the code is freely available on our Web site, at <http://puzzles.mit.edu/firefox/>.



# Chapter 7

## Conclusions and Future Work

We explore the notion of a puzzle only solvable by humans (POSH), and propose metrics on which good POSHes should be evaluated: consistency, entropy, fun, ease of generation, and ease of implementation. With this framework in mind, we implemented and got feedback on seven different POSHes: Word Association, Word Grouping, Draw-a-Noun, Draw-a-Phrase, Inkblot, Face Recognition, and CAPTCHA. Our preliminary results suggest that something like Word Association, Inkblot, and Draw-a-Noun, along with traditional CAPTCHA, have potential to be strong POSHes, though we suggest that more extensive user study is in order for each of them.

In addition, we show a security application for POSHes, and provide a concrete implementation of the CHS scheme, using a Word Association POSH. In so doing, we end up with an interesting finding: the puzzle selection provides a quick confirmation to the user that he or she entered the password correctly. In fact, this idea can be extended fairly straightforwardly in the online case. If the server uses an HMAC-SHA1 for `Expand`, the CHS scheme can provide mutual authentication: after entering the password, only the real server can compute which POSHes need to be presented to the user. If the user does not see his familiar POSHes, he knows that something is amiss. This is similar to SiteKey [Ban], but the puzzles are computed on a secret password, rather than an easily-guessable username.

However, more work remains to be done. For example, the CHS scheme offers no security against a few common threats: it is vulnerable to a keylogger/replay attack,

as well as to shoulder surfers. Solutions to these problems with the system would go a long way in increasing the practicality of its use.

Furthermore, while we have provided first steps at proposing and analyzing POSHes, many other candidates need to be explored, and more precise analysis needs to be undertaken. Some POSH suggestions are provided in [CHS06], but additional work in this area is required. One set of promising candidates appears to be *games*: a POSH derived from a game has the advantage that it is essentially guaranteed to be fun, since people already “solve” them for entertainment.

That said, more work is also needed to discover exactly how much entropy is actually present in each POSH, as a way of determining how vulnerable they are to guessing attacks.

Additionally, such studies need to be done in settings more regulated than a Web site, where users can be forced to re-solve puzzles on a regular basis. Practicing the same puzzle several times after it is first selected would also help address the issue of “drift” – where users answer with some response  $a$  the first time, forget it, and then substitute a different response  $b$  in all subsequent answers.

We end with the following suggestion: ways to improve the user experience should be paramount. Fundamentally, security comes down to user cooperation, and if the system is too tedious, people will not use it, regardless of the benefits it may potentially provide.

# Bibliography

- [Ban] Bank of America. How Bank of America SiteKey Works for Online Banking Security. <http://www.bankofamerica.com/privacy/sitekey/>.
- [BM92] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *SP '92: Proceedings of the 1992 IEEE Symposium on Security and Privacy*, page 72, Washington, DC, USA, 1992. IEEE Computer Society.
- [BM93] Steven M. Bellovin and Michael Merritt. Augmented encrypted key exchange: A password-based protocol secure against dictionary attacks and password file compromise. In *ACM Conference on Computer and Communications Security*, pages 244–250, 1993.
- [BPR00] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In Bart Preneel, editor, *Advances in Cryptology — Eurocrypt '00*, volume 1807 of *Lecture Notes in Computer Science*, page 139, 2000.
- [Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.
- [CD08] Ran Canetti and Ronny Ramzi Dakdouk. Obfuscating point functions with multibit output. In Nigel P. Smart, editor, *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 489–508. Springer, 2008.
- [CFB01] Allison L. Coates, Richard J. Fateman, and Henry S. Baird. Pessimism: A reverse turing test. In *ICDAR*, pages 1154–1158. IEEE Computer Society, 2001.
- [CHK<sup>+</sup>05] Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. Universally composable password-based key exchange. In *EUROCRYPT*, pages 404–421, 2005.
- [CHS06] Ran Canetti, Shai Halevi, and Michael Steiner. Mitigating dictionary attacks on password-protected local storage. In *CRYPTO*, pages 160–179, 2006.
- [Cor91] Fernando J. Corbató. On building systems that will fail. *Commun. ACM*, 34(9):72–81, 1991.

- [Dah08] Waseem Daher. Prompt for the master password via a XUL window. <https://bugzilla.mozilla.org/attachment.cgi?id=314766&action=edit>, April 2008. Patch for Firefox.
- [DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.
- [DMR04] Darren Davis, Fabian Monrose, and Michael K. Reiter. On user choice in graphical password schemes. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 11–11, Berkeley, CA, USA, 2004. USENIX Association.
- [DN93] Cynthia Dwork and Moni Naor. Pricing via Processing or Combatting Junk Mail. In *CRYPTO '92: Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, pages 139–147, London, UK, 1993. Springer-Verlag.
- [DP00] Rachna Dhamija and Adrian Perrig. Déjà vu: A user study, using images for authentication. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.
- [EKW74] Arthur Evans, Jr., William Kantrowitz, and Edwin Weiss. A user authentication scheme not requiring secrecy in the computer. *Commun. ACM*, 17(8):437–442, 1974.
- [Eng07] Daniel Engel. Pictionary (NES) FAQ by DEngel. <http://www.gamefaqs.com/console/nes/file/587510/38806>, April 2007. Section (E): Answer List.
- [Fou01] Mozilla Foundation. Glossary: Software Security Device. [http://www.mozilla.org/projects/security/pki/psm/help\\_21/glossary.html#software\\_security\\_device](http://www.mozilla.org/projects/security/pki/psm/help_21/glossary.html#software_security_device), August 2001.
- [God02] Philip Brighten Godfrey. Text-based CAPTCHA algorithms. In *First Workshop on Human Interactive Proofs*. Unpublished Manuscript, 2002. [http://www.aladdin.cs.cmu.edu/hips/events/abs/godfreyb\\_abstract.pdf](http://www.aladdin.cs.cmu.edu/hips/events/abs/godfreyb_abstract.pdf).
- [Hal01] H. Tracy Hall. Tracy Hall's Amazing Inkblot Generator. <http://math.berkeley.edu/~hthall/ink.blots/>, January 2001.
- [JMM<sup>+</sup>99] Ian Jermyn, Alain Mayer, Fabian Monrose, Michael K. Reiter, and Aviel D. Rubin. The design and analysis of graphical passwords. In *SSYM'99: Proceedings of the 8th conference on USENIX Security Symposium*, Berkeley, CA, USA, 1999. USENIX Association.
- [Kal00] Burt Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational), September 2000.

- [KLS02] Greg Kochanski, Daniel Lopresti, and Chilin Shih. A reverse turing test using speech. In *International Conferences on Spoken Language Processing*, pages 1357–1360, Denver, Colorado, 2002.
- [LABB01] Mark D. Lillibridge, Martin Abadi, Krishna Bharat, and Andrei Z. Broder. Method for selectively restricting access to computer systems, February 27 2001. U. S. Patent No. 6,195,698.
- [LS05] Shujun Li and Heung-Yeung Shum. Secure Human-Computer Identification (Interface) Systems against Peeping Attacks: SecHCI. Cryptology ePrint Archive, Report 2005/268, 2005. <http://eprint.iacr.org/>.
- [MT79] Robert H. Morris and Ken Thompson. Password security: a case history. *Commun. ACM*, 22(11):594–597, 1979.
- [Nao96] Moni Naor. Verification of a human in the loop or Identification via the Turing Test. [http://www.wisdom.weizmann.ac.il/~naor/PAPERS/human\\_abs.html](http://www.wisdom.weizmann.ac.il/~naor/PAPERS/human_abs.html), 1996.
- [Nie08] Han-Kwang Nienhuys. HKCaptcha – yet another PHP Captcha implementation. <http://www.lagom.nl/linux/hkcaptcha/>, February 2008.
- [PS02] Benny Pinkas and Tomas Sander. Securing passwords against dictionary attacks. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 161–170, New York, NY, USA, 2002. ACM Press.
- [SBOR06] Pawan Sinha, Benjamen Balas, Yuri Ostrovsky, and Richard Russell. Face Recognition by Humans: Nineteen Results all Computer Vision Researchers Should Know About. In *Proceedings of the IEEE*, volume 94, pages 1948–1962, 2006.
- [Sha] Open Source Shakespeare. Shakespeare text statistics. <http://www.opensourceshakespeare.org/stats/>.
- [SS04] Adam Stubblefield and Dan Simon. Inkblot authentication. Technical Report MSR-TR-2004-85, Microsoft Research, 2004.
- [TvO04] Julie Thorpe and P. C. van Oorschot. Graphical dictionaries and the memorable space of graphical passwords. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.
- [vA07] Luis von Ahn. reCAPTCHA. <http://www.recaptcha.net>, 2007.
- [vABL04] Luis von Ahn, Manuel Blum, and John Langford. Telling humans and computers apart automatically. *Communications of ACM*, 47:56–60, 2004.