PAPER *Special Section on Cryptography and Information Security*

Mitigating Dictionary Attacks with Text-Graphics Character CAPTCHAs

Chanathip NAMPREMPRE[†a)], ***Member and*** **Matthew DAILEY**[††b)], ***Nonmember***

SUMMARY We propose a new construct, the Text-Graphics Character (TGC) CAPTCHA, for preventing dictionary attacks against password authentication systems allowing remote access via dumb terminals. Password authentication is commonly used for computer access control. But password authentication systems are prone to dictionary attacks, in which attackers repeatedly attempt to gain access using the entries in a list of frequently-used passwords. CAPTCHAs (Completely Automated Public Turing tests to tell Computers and Humans Apart) are currently being used to prevent automated "bots" from registering for email accounts. They have also been suggested as a means for preventing dictionary attacks. However, current CAPTCHAs are unsuitable for text-based remote access. TGC CAPTCHAs fill this gap. In this paper, we define two TGC CAPTCHAs and incorporate one of them in a prototype based on the SSH (Secure Shell) protocol suite. We also prove that, if a TGC CAPTCHA is easy for humans and hard for machines, then the resulting CAPTCHA is secure. We provide empirical evidence that our TGC CAPTCHAs are indeed easy for humans and hard for machines through a series of experiments. We believe that a system exploiting a TGC CAPTCHA will not only help improve the security of servers allowing remote terminal access, but also encourage a healthy spirit of competition in the fields of pattern recognition, computer graphics, and psychology.[*]

key words: *Reverse Turing Test, CAPTCHA, Secure Shell, Password Authentication*

1. Introduction

Password authentication is one of the most common building blocks in implementing access control. Each user has a relatively short sequence of characters commonly referred to as a *password*. To gain access, the user provides his/her password to the system. Access is granted if the password is correct; it is denied otherwise.

A common attack against password authentication systems is the *dictionary attack*. An attacker can write a program that, trying to imitate a legitimate user, repeatedly tries different passwords, say from a dictionary, until it finds one that works.

There are several well-known ways to cope with dictionary attacks. For example, the system can deny access for the user in question after some number of tries, a technique known as account locking. However, this invites a denial of service attack: an attacker can lock anyone out of the system by submitting a sequence of incorrect passwords on behalf of the victim. Other solutions also have their own shortcomings [1].

In this paper, we present an alternative defense against dictionary attacks. The idea is to make it harder for automated programs to mount dictionary attacks by requiring an attacking program to pass a test that is easy for humans but is hard (in terms of accuracy and/or compute time) for computer programs. A construct with this property is called a CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) [2]. In particular, if there exists a program that can pass a CAPTCHA with high probability, then that program can be used to solve a hard AI problem. CAPTCHAs are already in use in some systems that benefit from distinguishing between humans and "bots" [3]. Recently, they have also been suggested as a means for deterring dictionary attacks in password authentication systems [1].

Some CAPTCHA suitable for password authentication systems displays a degraded image of a word to the human, who then responds by typing the word he or she sees. Another CAPTCHA uses a sound clip instead of an image. However, these CAPTCHAs are not suitable for systems that allow remote access via consoles or dumb terminal programs. Our goal is to make it possible for such minimal systems to obtain the same benefits from CAPTCHA-assisted password authentication as systems with graphical displays and/or speakers.

To this end, we propose new CAPTCHAs based on Text-Graphics Characters. We call such a construct a *TGC CAPTCHA*. A *text-graphics character* is an image of a character rendered on a text-only screen, namely a screen in which ASCII characters are used to represent pixels. Unfortunately, images rendered on a text-only screen are necessarily low resolution. This limits the number of characters that can be put on a single screen without hindering recognition. In fact, we find that on an 80x24 screen, it is unreasonable to display more than one or two distorted English characters. Without the

Manuscript received March 23, 2006.
Manuscript revised July 5, 2006.
Final manuscript received January 1, 2007.

[†]The author is with the Electrical Engineering Department, Faculty of Engineering, Thammasat University.
[††]The author is with the Computer Science and Information Management Program, Asian Institute of Technology.
a) E-mail: nchanath@engr.tu.ac.th
b) E-mail: mdailey@ait.ac.th

[*]Based on "A Text-Graphics Character CAPTCHA for Password Authentication," by M.N. Dailey and C. Namprempre, which appeared in the TENCON 2004 Conference Proceedings, ©2004, IEEE.

contextual information afforded when complete words are displayed, humans have a harder time recognizing what is on each screen. Additionally, neither color nor grayscale are necessarily available to help provide visual clues. These limitations together make it difficult to generate text-graphics characters that are easy for humans but hard for computers to recognize.

Contributions. First, we precisely define what a TGC CAPTCHA is and prove that, if it is easy for humans and hard for machines, then it is indeed a secure CAPTCHA according to the definitions in [2]. Specifically, we offer a reductionist proof to show that, if a particular class of problems believed to be hard is in fact a hard AI problem as defined in [2], then our TGC CAPTCHAs are secure. As such, the term "prove" here is used in the same sense as that in the area of *provable security* [4], namely that the security of a construct, i.e. the CAPTCHA, is proved based on the security of a primitive, i.e. the corresponding hard AI problem.

Second, we implement two TGC CAPTCHAs. Then, we incorporate one of them into the user authentication mechanism of the popular Secure Shell (SSH) protocol suite, thus hardening SSH servers against dictionary attacks. The prototype is available as a source code patch for OpenSSH 3.6.1 at [5].

Finally, we provide empirical evidence that both of our TGC CAPTCHAs are easy for humans and relatively hard for machines by running CAPTCHA tests against human subjects and an Optical Character Recognition (OCR) program [6], respectively. The results are encouraging: on a character-by-character level, humans achieve more than 95% accuracy on both TGC CAPTCHAs after two practice trials, whereas the OCR program's accuracy is less than 35% for both TGC CAPTCHAs. We believe that improving the OCR system's accuracy to approach that of humans will require more effort on the part of OCR system designers, and in any case, attackers using OCR systems to mount automated dictionary attacks will require a substantial amount of total compute time.

TGC CAPTCHA with Visual Noise. The first TGC CAPTCHA presents a sequence of k distorted characters, one at a time, to the user and accepts only correct responses ($k = 8$ in our implementation). To make the test easy for humans, we use only uppercase English characters, excluding the characters 'O' and 'D'. For each character, we pick n_d "distracters" and apply several operations to both the character and the distracters before laying the former over the latter. The operations are scaling, rotation, translation, and "row sliding," an operation involving horizontally moving each pixel in incremental steps, row-wise. The characters, the distracters, the operations, and all of the parameters are chosen at random from configurable ranges. The resulting image is then displayed to the user.

TGC CAPTCHA in Random Figlet Fonts. The second TGC CAPTCHA presents a sequence of k characters (again, $k = 8$), each displayed as a *figlet*, a big character displayed using a collection of ordinary ASCII characters [7]. For each figlet character, we select a random *figlet font* for the character. (See Figure 5 for examples of figlets in different fonts.) We use both upper and lower case here but do not ask the user to distinguish between a character in the upper and that in the lower case. To make the test easy for humans, we exclude 'I', 'L', 'O', and 'D' in both upper and lower case.

The Prototype in SSH. We add the TGC CAPTCHA with visual noise to SSH. All our modifications are compliant with SSH specifications [8]. The server program, when configured with CAPTCHA support, informs the client that CAPTCHA-based authentication is a valid method. The client, when configured with CAPTCHA support, requests CAPTCHA authentication. The server then transmits a sequence of transformed images of characters to the client. The client displays the received images one by one and collects responses from the user before sending them all back to the server with the user's password (thus minimizing the impact of network delays). The server grants access to the user only if he/she *both* passes the CAPTCHA test *and* enters the correct password. The server denies access otherwise.

Features of TGC CAPTCHAs and the Prototype. TGC CAPTCHAs provide a defense against online dictionary attacks without resorting to other countermeasures, e.g. account locking, with well-known disadvantages. Also, as discussed in [1], even if solving the underlying AI problem, in this case the problem of recognizing distorted text-graphics characters, turns out to be easy for OCR systems, it would still be useful to use a TGC CAPTCHA in password authentication systems. The reason is that an attacker mounting an online dictionary attack still needs *both* to solve the CAPTCHA *and* to find the password in order to login successfully. In this situation, our approach reduces to *pricing via processing*, a paradigm originally proposed to combat senders of bulk junk email ("spam") [9]. In order to mount a dictionary attack, the adversary must perform a moderately complex computational task on each password authentication attempt. Thus, even if the attacker only spends a few compute cycles on each trial, the cumulative effect becomes significant over the many trials required for a dictionary attack.

We stress here that the use of a TGC CAPTCHA in password authentication is only one possible application. Like other CAPTCHAs, TGC CAPTCHAs can be used in any application in which it is useful to distinguish human users from bots [2,3]. Furthermore, TGC CAPTCHAs extend these benefits to applications with console-based interfaces. For example, it can help prevent bots from signing up for free email accounts or help conduct online polls via, say, text-based web browsers. Search engines that disallow bots by requiring graphical CAPTCHAs to be recognized will no longer end up automatically rejecting legitimate users who access the sites via a text-based interface, if a TGC CAPTCHA is made available to those users.

Related Work. von Ahn et al. [2] were the first to propose the concept of CAPTCHAs. They formally defined the CAPTCHA construct and its security notion and specified two classes of AI problems. Our TGC CAPTCHAs are instances of their second problem class.

Pinkas and Sanders proposed the use of *Reverse Turing Tests* (RTTs), constructs similar to CAPTCHAs, to cope with dictionary attacks [1]. Their focus was on usability and scalability: they wanted to ensure that the addition of RTTs did not require dramatic changes in the users' behavior and level of effort. They proposed a concrete login protocol similar to our prototype. We mention only some of the differences here: (1) they ask for the password before asking the user to solve the RTT; (2) they require the user to solve the RTT only some fraction of the time; and (3) they use a persistent data structure, namely a web cookie. Xu et al. proposed the use of character recognition to separate humans from machines [10]. However, they did not assume authenticated, replay-resilient channels. Consequently, the communication had to be protected via message authentication codes and serial numbers, timestamps, or state information. In contrast, our CAPTCHA challenges and responses are sent over SSH channels which are already encrypted, authenticated, and replay-resilient [11]. This dramatically simplifies our protocol.

Notation. Let k be a positive integer. We denote by $x_1, \ldots, x_k \overset{\$}{\leftarrow} X$ the act of sampling each element x_i uniformly and independently from the set X. We write $C \rightarrow S\colon msg$ to denote a move in a protocol where the party C sends a message msg to the party S.

2. Our Text-Graphics Character CAPTCHAs

In this section, we describe the TGC CAPTCHAs that we have implemented following the formalization in [2].

2.1 TGC CAPTCHA with Visual Noise

Let $\mathcal{I}$ be a set of images of all upper case English characters, $\mathcal{T}$ be a set of transformations on images, λ be the map from an image of a character to the (ASCII ID of) the character portrayed in the image, and τ and k be the security parameters. The TGC CAPTCHA TGC_1 is a tuple $(\mathcal{I}, \mathcal{T}, \lambda, \tau, k)$ defining the test shown in Figure 1. First, the verifier (i.e. server) draws $i_1, \ldots, i_k \overset{\$}{\leftarrow} \mathcal{I} - \{\text{'O'}, \text{'D'}\}$ and $t_1, \ldots, t_k \overset{\$}{\leftarrow} \mathcal{T}$.

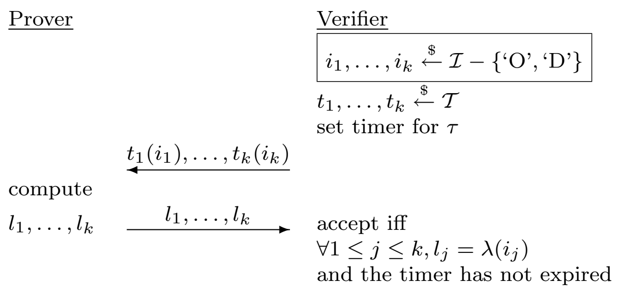
Prover
Verifier
$\boxed{i_1, \ldots, i_k \overset{\$}{\leftarrow} \mathcal{I} - \{\text{'O'}, \text{'D'}\}}$
$t_1, \ldots, t_k \overset{\$}{\leftarrow} \mathcal{T}$
set timer for τ
$t_1(i_1), \ldots, t_k(i_k)$
compute
$l_1, \ldots, l_k$
$l_1, \ldots, l_k$
accept iff
$\forall 1 \le j \le k, l_j = \lambda(i_j)$
and the timer has not expired

Fig. 1 Text-Graphics Character CAPTCHAs. An instance of a TGC CAPTCHA is $\mathsf{TGC} = (\mathcal{I}, \mathcal{T}, \lambda, \tau, k)$. The protocol shown is for TGC_1. For TGC_2, the set $\mathcal{I}$ includes both upper and lower case English letters, and the boxed text is replaced by $i_1, \ldots, i_k \overset{\$}{\leftarrow} \mathcal{I} - \{\text{'I'}, \text{'L'}, \text{'O'}, \text{'D'}\}$.

Fig. 2 Our choice for the reference image set $\mathcal{I}$ for TGC_1. We use only uppercase English characters but omit the characters 'O' and 'D' because they are hard to distinguish when distorted.

Then, it sends to the prover (i.e. user) the transformed images $t_1(i_1), \ldots, t_k(i_k)$ and sets the timer for τ. The prover responds with the labels $l_1, \ldots, l_k$, each of which is (the ASCII ID of) a character in the English alphabet. The verifier accepts if $l_j = \lambda(i_j)$ for all $1 \le j \le k$ and if the timer has not expired. It rejects otherwise.

We describe here our choices for TGC_1 for the sets $\mathcal{I}$ and $\mathcal{T}$. The reference image for each character, from the standard X Window System "9x15" font, is shown in Figure 2. We use all of the uppercase English characters except 'O' and 'D' which are practically indistinguishable when distorted. The transformation process involves the following steps. First, n_d distracters are chosen uniformly with replacement from the set of all distracter images. In our implementation, we use $n_d = 5$ samples from a set of 26 9x15 bitmaps that share some features with English letters but are easily classified as non-letters by humans. We perform the following operations:

1. Map to ASCII: The 9x15 bitmap is mapped to a 9x15 array of ASCII characters, also known as a *charmap*. White pixels are mapped to the space (' ') character, and black pixels are mapped to the asterisk ('*') character.
2. Scale: The charmap is scaled by a random factor between 1.3 and 1.7.
3. Rotate: The charmap is rotated by a random angle between -20 and 20 degrees.
4. Translate: The charmap is translated to a random location on the $n_c \times n_r$ screen with the constraint that the entire character must still be visible.
5. Row Slide: Each row of the charmap is optionally slid left or right relative to the row above. We shift left with probability 0.33, right with probability 0.33, and otherwise do not shift.

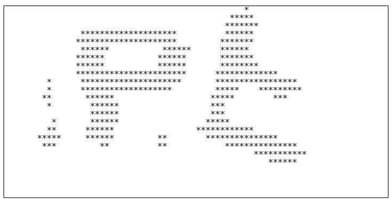
Fig. 3 Screen shot of a rendered TGC CAPTCHA character corresponding to the character 'P' for TGC_1.

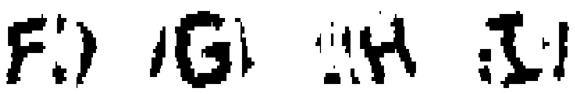
Fig. 4 Example TGC CAPTCHA characters for TGC_1.

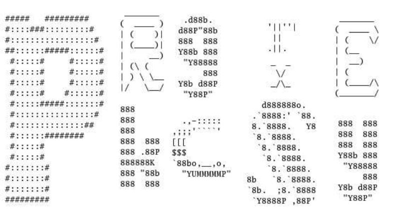
Fig. 5 Example TGC CAPTCHA characters for TGC_2.

In our implementation, all of the random samples are from a uniform distribution and rely on the standard C library `random()` function (seeded with the system time). Finally, the transformed target and distracter images are overlaid as follows. Each overlay is given an opaque whitespace boundary, and the target image is laid down last, to ensure that the target stands out clearly from the background. Figure 3 shows an example of a transformed character as it would be rendered on the screen. Figure 4 shows at a lower resolution examples of how some characters may be transformed.

We believe that the set of transformations that we have described gives sufficient variability in the output images to make recognition fairly difficult for machines while preserving the necessary property that it is easy for humans. Experienced users (e.g. the authors) consistently classify the characters with 100% accuracy, and as we shall see in Experiment 1A below, even naive users perform well enough to make our construction practical for deployment in real systems.

2.2 TGC CAPTCHA in Random Figlet Fonts

The TGC CAPTCHA in Random Figlet Fonts $\mathsf{TGC}_2 = (\mathcal{I}, \mathcal{T}, \lambda, \tau, k)$ is similar to TGC_1 shown in Figure 1. The differences between the two are in the choices of the sets $\mathcal{I}$ and $\mathcal{T}$. Specifically, $\mathcal{I}$ is the set of English characters in both upper and lower case excluding 'I', 'L', 'O', and 'D'. The set $\mathcal{T}$ contains the figlet fonts `basic`, `big`, `block`, `broadway`, `colossal`, `cosmic`, `cybermedium`, `doh`, `doom`, `dotmatrix`, `epic`, `fender`, `nancyj`, `ogre`, `pebbles`, `puffy`, `roman`, `rounded`, `starwars`, `stop`, `univers`, and `whimsy` [7]. A few examples are shown in Figure 5. Computing $t(i)$ where $t \in \mathcal{T}$ and $i \in \mathcal{I}$ yields character i in the figlet font t.

The fonts that we choose give sufficient variability in the output images to make recognition fairly difficult for machines while remaining relatively easy for humans. Experiment 1B below shows that even naive users perform well against this TGC CAPTCHA.

3. CAPTCHA-Augmented Password Authentication

We have found that TGC CAPTCHAs can complement password-based authentication systems. In this section, we give an overview of how a TGC CAPTCHA can be applied to user authentication, then describe an implementation of the general approach as part of the SSH user authentication protocol.

3.1 Overview

A TGC CAPTCHA can complement any interactive password authentication scheme, so long as a sufficiently large text console is available for displaying the CAPTCHA. The type of password-based authentication protocols we focus on is that described in [1]. In particular, a user trying to gain access to a system is challenged by a CAPTCHA test and asked to enter his/her password. If the user both passes the CAPTCHA test and correctly enters the password, then access is granted. Otherwise, access is denied.

To ensure that a denied request gives no indication of whether the user has failed the CAPTCHA or has entered an incorrect password, the user must not receive any feedback until after both solving the CAPTCHA and entering a password. We also assume that the communication channel used during protocol execution is encrypted, authenticated, and replay-resilient (as is the case, for example, in SSH).

3.2 Putting CAPTCHA to Use in SSH

We have implemented a prototype TGC CAPTCHA password authentication method compatible with the SSH user authentication protocol [8]. As a concrete example, we base our implementation on the TGC CAPTCHA TGC_1 and OpenSSH 3.6.1. However, the method could just as easily be based on TGC_2 and/or be incorporated into any SSH-compliant client or server. The latter is so because SSH was specifically designed to allow new user authentication methods to be added in a modular fashion.

Password authentication in SSH. The SSH user authentication protocol [11] provides several authentication methods, but we focus on the password-based method here. The SSH password authentication

$S \rightarrow C$: available authentication methods
C: get user's password
$C \rightarrow S$: `SSH_MSG_USERAUTH_REQUEST,`
`user name = <uname>, service name = <sname>,`
`method name = password, change = FALSE,`
`password = <password>`
S: check password
$S \rightarrow C$: `SSH_USERAUTH_SUCCESS`

Fig. 6 A successful SSH authentication protocol with the standard `password` method. The client C and the server S are assumed to have established a transport-layer session beforehand so that integrity and confidentiality are guaranteed during the protocol. We denote by `<var>` a variable `var` whose value depends on the context of the protocol execution.

method is shown schematically in Figure 6. We emphasize that the user authentication protocol runs on top of the SSH transport protocol, which provides integrity and confidentiality [11]. Here we describe the essential steps in a password authentication exchange; for details, please refer to the specification [8].

The protocol begins with the SSH server telling the client which authentication methods are available. The SSH client is then free to request authentication using any of the available methods.

To request password-based authentication, the client sends a `SSH_MSG_USERAUTH_REQUEST` message (type 50) to the server. This message contains one byte for the message type followed by a user name (a UTF-8 encoded string), a service name (a US-ASCII encoded string), and a method name (also a US-ASCII string). When the method name is `password`, two more fields follow: a one-byte boolean flag indicating whether this request is a password change request, and a cleartext UTF-8 encoded password. The former is used to indicate when the request is a response to a `SSH_MSG_USERAUTH_PASSWD_CHANGEREQ` from the server, but it is ignored in OpenSSH 3.6.1, which does not implement password change requests. The latter specifies the password.

On receipt of the `SSH_MSG_USERAUTH_REQUEST` message, the server initiates one or more challenge-response interactions, then under normal circumstances, it will eventually respond with `SSH_MSG_USERAUTH_SUCCESS` (type 52) or `SSH_MSG_USERAUTH_FAILURE` (type 51). In the case of success, the user authentication protocol is complete. In the case of failure, the client is free to re-attempt authentication using the same or a different method. After some (configurable) number of failed authentication attempts, the server terminates the session.

This standard password authentication method only involves a single exchange of messages between the client and server. But the SSH standard also allows for the implementation of authentication methods that involve multiple exchanges. We exploit this capability to add CAPTCHAs to the existing password authentication method.

$S \rightarrow C$: available authentication methods
$C \rightarrow S$: `SSH_MSG_USERAUTH_REQUEST,`
`user name = <uname>, service name = <sname>,`
`method name = captcha-password`
S: generate `<captchas>`
$S \rightarrow C$: `SSH_MSG_CAPTCHA, number of columns = <nc>,`
`number of rows = <nr>, number of letters = <k>,`
`data = <captchas>`
C: get user's password and responses to `<captchas>`
$C \rightarrow S$: `SSH_MSG_USERAUTH_CAPTCHA_ RESPONSE,`
`response = <captcha response>, change = FALSE,`
`password = <password>`
S: check password and responses to `<captchas>`
$S \rightarrow C$: `SSH_USERAUTH_SUCCESS`

Fig. 7 A successful SSH authentication protocol with the CAPTCHA-augmented `captcha-password` method. The client C and the server S are assumed to have established a transport-layer session beforehand so that integrity and confidentiality are guaranteed during the protocol. We denote by `<var>` a variable `var` whose value depends on the context of the protocol execution.

Adding CAPTCHAs to SSH password authentication. We implemented a new SSH authentication method, `captcha-password`, that incorporates both the password authentication request and a CAPTCHA challenge-response exchange. Here we briefly describe the protocol flow, which is illustrated in Figure 7.

1. As always, the SSH server sends the client a list of acceptable authentication methods. If the server is configured with the CAPTCHA authentication method, the string `captcha-password` is sent as one of the acceptable methods.

2. The client sends a `SSH_MSG_USERAUTH_REQUEST` message to the SSH server with the usual username and service fields, but sets the method field to `captcha-password`. The `captcha-password` method does not add any method-specific fields to the request message.

3. The server creates a sequence of TGC CAPTCHA characters for a random word. The number of letters in the word is configurable.

4. The server creates a `SSH_MSG_CAPTCHA` (type 62) message containing the CAPTCHA images. The first byte is the message type. The next 12 bytes specify the number of columns (n_c), rows (n_r), and letters (k) in the CAPTCHA. Our implementation uses $n_c = 80, n_r = 24, k = 8$ by default. The remainder of the message is the payload: $n_c \times n_r \times k$ US-ASCII encoded characters to be displayed by the client. Once the message is created, it is encrypted, packetized, and sent to the client by the transport layer.

5. On receipt, the client decodes the `SSH_MSG_CAPTCHA` message, displays each text-graphics image in sequence, and records the user's responses. It then prompts the user to enter her password.

6. The client creates and sends a `SSH_MSG_USERAUTH_CAPTCHA_RESPONSE` (type 63) message containing the password and CAPTCHA response. The first byte is

the message type. This is followed by a string specifying the user's response to the CAPTCHA, the one-byte password change request field, and the user's password response. Any characters in the CAPTCHA response string outside the range '`a`'-'`z`' are first converted to a space (' ') character.

`7.` On receipt of the CAPTCHA response message, the server first checks the correctness of the user response. If incorrect, the server sends a `SSH_MSG_USERAUTH_FAILURE` message and aborts the authentication attempt. If correct, the server checks the user's password as usual in the `password` authentication method. Again, if the password is incorrect, the server sends a `SSH_MSG_USERAUTH_FAILURE` message and aborts the authentication attempt. If the password is correct, the server sends the `SSH_MSG_USERAUTH_SUCCESS` message and proceeds to service the user's request. In the case of failure, the client is free to re-attempt authentication until the server decides to close the connection.

The actual changes required to implement our authentication method on top of OpenSSH 3.6.1 are minimal. The modified `sshd` server now requires the C math libraries for the graphics transforms, and the `ssh` client now requires the `curses` library to display TGC CAPTCHAs on text terminals. Source code patches relative to OpenSSH 3.6.1 are available at [5].

4. Theoretical Result

We instantiate a problem from the class of problems $P2$ defined in [2] and reduce the security of a TGC CAPTCHA to the hardness of the problem. We describe roughly the definition of $P2$ and some key terms. See [2] for more precise definitions.

The family of AI problems $P2$ is indexed by the distributions of images $\mathcal{I}$ and $\mathcal{T}$ and by the solution λ which is a map of images to their corresponding labels. Intuitively, it is the following problem: given a transformed image $t(i)$ where $i \in \mathcal{I}$ and $t \in \mathcal{T}$, find the label $\lambda(i)$. A problem is (δ, τ)*-hard* if no current program can solve it with probability at least δ in time at most τ. A test is (α, β)*-human executable* if at least an α portion of the human population can pass it with at least β success probability. An (α, β, η)*-CAPTCHA with respect to a* (δ, τ)*-hard problem* P is a test that is (α, β)-human executable and has the property that, if an algorithm B passes it with a success probability of at least η, then B can be used to solve the problem P with probability at least δ in time at most τ.[†]

In discussion, we say that a problem is "hard" if it is (δ, τ)-hard for any program running in "reasonable" time τ with "reasonable" success probability δ. We say that a CAPTCHA is "secure" if it is an (α, β, η)-CAPTCHA with respect to a hard problem for "reasonable" value of η. Intuitively, the following theorem states that each CAPTCHA $\mathsf{TGC} = (\mathcal{I}, \mathcal{T}, \lambda, \tau, k)$ that we define in Section 2 is secure assuming that the underlying problem $P2_{\mathcal{I},\mathcal{T},\lambda}$ is hard.

Theorem 4.1: Let k be the security parameter, and let $\mathcal{I}, \mathcal{T}, \lambda$ be as previously defined in Section 2. Let $\delta, \tau, \alpha, \beta$ be non-negative real numbers. Assume that $\mathsf{TGC} = (\mathcal{I}, \mathcal{T}, \lambda, \tau, k)$ is (α, β)-human executable. If $P2_{\mathcal{I},\mathcal{T},\lambda}$ is $(\delta, \tau + O(k))$-hard, then TGC is a (α, β, δ)-CAPTCHA with respect to $P2_{\mathcal{I},\mathcal{T},\lambda}$.

Proof: Given an algorithm B that has a success probability of at least δ over TGC in time τ, we construct an algorithm A that is a $(\delta, \tau + O(k))$ solution to $P2_{\mathcal{I},\mathcal{T},\lambda}$. This proves the theorem.

On input a transformed image t^*, the algorithm A simply draws $g \overset{\$}{\leftarrow} \{1, \ldots, k\}$ and sets $w_g = t^*$. Then, for all j such that $1 \leq j \leq k$ and $j \neq g$, it draws $i_j \overset{\$}{\leftarrow} \mathcal{I}$ and $t_j \overset{\$}{\leftarrow} \mathcal{T}$, then sets $w_j = t_j(i_j)$. Next, it submits $w_1, \ldots, w_k$ to B and obtains the answers $l_1, \ldots, l_k$ from B. Finally, it outputs l_g as its answer.

It is easy to see that A simulates B perfectly. Also, A runs in time taken by B plus time linear in k. Finally, if B has probability δ of getting all of its answers right, then A's probability of success is at least δ. ▌

5. Experimental Results

We performed two experiments to assess the difficulty of our TGC CAPTCHAs for humans and machines.

Experiment 1: Playing against humans. In this experiment, we set out to answer the question of whether our TGC CAPTCHAs are easy enough for humans to be practical complements to password authentication.

We ran two experiments, 1A and 1B, for TGC_1 and TGC_2, respectively. For each experiment, 20 naive subjects were recruited from the faculty, staff, and students of Thammasat University and the Asian Institute of Technology. All of the subjects were competent, though non-native, English speakers. The test equipment was one of several PCs running Linux or Mac OS X. The TGC CAPTCHA characters were always displayed in a standard terminal window.

Each subject participated in an individual session lasting approximately 5 minutes. We gave brief written instructions explaining that they would be asked to identify English characters. They were instructed to maximize their accuracy without regard to time. The instructions were followed by two practice trials with two different sequences of $k = 8$ characters displayed

[†]Our definition for CAPTCHA is slightly different than the original definition in [2] in that we make the dependence on the hard problem underlying the CAPTCHA explicit.

on a $n_c = 80$ by $n_r = 24$ screen. At the end of each trial, the subjects received feedback on whether their response was correct or incorrect. If incorrect, their response and the correct response were displayed.

Following the practice trials were 10 test trials with the same parameters. During the test trials, the subjects' responses and response times were recorded. (They were not told that their response times were being recorded, however.)

For Experiment 1A, we used only upper case English characters excluding 'O' and 'D' and displayed them on noisy screens. For Experiment 1B, we used both upper and lower case English characters excluding 'I', 'L', 'O', and 'D' and displayed them in the figlet fonts listed in Section 2.2. In both experiments, the subjects were instructed that they could type either upper case or lower case responses without penalty.

The subjects' average per-character accuracy p_h on the test trials was 0.960 for Experiment 1A and 0.965 for Experiment 1B. Their average word-level accuracy (the number of 8-letter TGC CAPTCHAs answered with 100% accuracy) was 0.765 for Experiment 1A and 0.780 for Experiment 1B. (Assuming independence and $p_h = 0.960$ for 1A and $p_h = 0.965$ for 1B, we would expect a word-level accuracy $(p_h)^k$ of 0.721 for 1A and 0.752 for 1B.)

The fact that naive users achieve such high accuracy rates justifies the use of TGC CAPTCHAs in live systems. Frequent users would very rapidly adapt to the statistics of the character set, achieving even higher accuracy rates.

Experiment 2: Playing against a machine. We ran two experiments, 2A and 2B, for TGC_1 and TGC_2, respectively. In each experiment, we sought to put an upper bound on the difficulty of each TGC CAPTCHA for machines. To this end, we employed an Optical Character Recognition (OCR) system as an adversary against our CAPTCHAs. We selected GOCR [6] because it is open-source, has an active developer community, and runs on a variety of platforms including the UNIX-like operating systems that ship SSH by default.

Using the same parameters as Experiments 1A and 1B, for both Experiment 2A and 2B, we generated 100 TGC CAPTCHAs of length 8, for a total of $2 \times 100 \times 8 = 1600$ text-graphics characters. We then converted each textual display into a bitmap. Each row and column of the bitmap corresponds to a row and column in the text display. We mapped the background text character to white and all other characters to black.

We then built the GOCR 0.39 program from source code using its default configuration, and fed each bitmap directly to the program. In both experiments, we gave GOCR the legal set of characters it should detect, i.e. the 24 characters 'A'–'Z' excluding 'D' and 'O' for Experiment 2A and the 44 characters 'A'–'Z' and 'a'–'z' excluding both upper and lower case versions of 'D', 'I', 'L', and 'O'. We call this the *Naive GOCR* adversary to emphasize that different configurations could in principle yield better adversaries. After running Naive GOCR on each image, we classified its response as correct or incorrect.

One important difference between our OCR setup and the human experiment is that the OCR system was not constrained to respond with one and only one character for a given image. As a result, on some images, the OCR system responded with no characters, and on some images, it responded with multiple characters. We therefore measured the system's accuracy with two different criteria in an attempt to put rough bounds on the system's accuracy. As a conservative criterion, we judged the OCR system's response correct when it responded with the expected English letter and no other letters. As a less conservative criterion, we judged the OCR system's response correct any time the desired English letter appears in the OCR system's output, even if other letters also appear.

For Experiment 2A, naive GOCR had a per-character accuracy p_m of 0.278 and 0.314 by the first (conservative) and second (loose) criterion, respectively. The word-level accuracy was 0 by both criteria. For Experiment 2B, without any extraneous noise in the image, naive GOCR had the same per-character accuracy p_m of 0.330 by both the strict and loose evaluation criteria. The word-level accuracy was 0.

We believe that p_m could be significantly improved by incorporating problem-specific knowledge of the TGC CAPTCHA algorithm. We leave this as an exercise for interested readers. Clearly, however, the results demonstrate that Naive GOCR is unsuitable for mounting dictionary attacks against TGC CAPTCHA-enabled password authentication systems.

6. Conclusion

In this paper, we have proposed a construct called the TGC CAPTCHA, a new CAPTCHA requiring only the most basic user interface equipment: a dumb terminal (or terminal emulation program) and a keyboard. A TGC CAPTCHA is therefore suitable for protecting password authentication systems from automated dictionary attacks, even in systems that are not equipped with graphical user interfaces.

A TGC CAPTCHA shows promise as a construct for improving the security of password authentication systems. As an example application, we have implemented and tested a new CAPTCHA-based password authentication method for the popular SSH protocol suite. We note, however, that TGC CAPTCHAs could be utilized for any application in which it is useful to distinguish human users from robots over a console-based interface.

We have implemented two TGC CAPTCHAs and have shown that they are relatively easy for humans but would be difficult for “Naive GOCR” adversaries. Of course, this only puts an *upper bound* on the difficulty of the problem. We have not proven that no adversary can do better. (After all, certain AI pundits believe that ALL tasks humans can perform today will be performed equally well by machines in the not-so-distant future!) However, we emphasize that, for a TGC CAPTCHA to be of use in securing password authentication systems against dictionary attacks, we do not *require* that it be necessarily more difficult for machines than humans. To the contrary, it is only necessary to force attackers to expend enough compute time to make attacks based on password searches impractical.

However, forcing attackers to expend additional compute time is not the only benefit of the CAPTCHA approach. Since the security of a CAPTCHA password authentication system increases as the gap between human and machine performance on the test widens, TGC CAPTCHAs can serve as a modest cross-disciplinary challenge in the fields of pattern recognition, system security, computer graphics, and even psychology. This is the approach championed by [2]. Through friendly competition, we hope to encourage not only new OCR algorithms, but also a better understanding of the strengths and weaknesses of the human visual system relative to the best present-day machine vision systems.

7. Future Work

In this paper, we have demonstrated the efficacy of TGC CAPTCHAs as a tool to improve the security of console-based network applications. In forthcoming work, we plan the following improvements. First, we would like to improve the performance of the Naive GOCR adversary as it is currently somewhat weak. For example, rather than searching for arbitrary text in the CAPTCHA window, the adversary could be constrained to perform a 24-way or 44-way forced choice, thus improving its chance of success. Likewise, as the adversary improves, we may need to improve the CAPTCHAs in turn, making them harder for machines without making it more difficult for humans. Finally, our `captcha-password` authentication method for SSH needs to be generalized before it will be acceptable as an Internet standard.

Acknowledgments

We thank the faculty, staff, and students of Thammasat University and the Asian Institute of Technology for their help with Experiments 1A and 1B. We also thank Wachira Promsaka Na Sakolnakorn for his helpful comments. The first author is supported by the Thailand Research Fund MRG49.

References

[1] B. Pinkas and T. Sander, “Securing passwords against dictionary attacks,” Proc. of the 9th CCS, ed. V. Atluri, pp.161–170, ACM Press, Nov. 2002.

[2] L. von Ahn, M. Blum, N.J. Hopper, and J. Langford, “CAPTCHA: Using hard AI problems for security,” EUROCRYPT 2003, ed. E. Biham, LNCS, vol.2656, pp.294–311, Springer-Verlag, May 2003.

[3] The CAPTCHA Project, `http://www.captcha.net`.

[4] M. Bellare, “Practice-oriented provable security,” Lectures on Data Security: Modern Cryptology in Theory and Practice, ed. I. Damgård, LNCS, vol.1561, pp.1–15, Springer-Verlag, 1998.

[5] M. Dailey and C. Namprempre. `http://www.cs.ait.ac.th/~mdailey/captcha/`.

[6] J. Schulenburg *et al.*, “GOCR: open-source character recognition, version 0.39,” 2004. Available at `http://jocr.sourceforge.net/index.html`.

[7] G. Chappell, I. Chai, and C. Keet, “FIGlet version 2.2.2,” 2005. Available at `http://www.figlet.org`.

[8] T. Ylonen, “The secure shell (SSH) authentication protocol.” IETF RFC 4252, Jan. 2006.

[9] C. Dwork and M. Naor, “Pricing via processing or combatting junk mail,” CRYPTO 1992, ed. E. Brickell, LNCS, vol.740, pp.139–147, Springer-Verlag, Aug. 1992.

[10] J. Xu, R. Lipton, I. Essa, M. Sung, and Y. Zhu, “Mandatory human participation: A new authentication scheme for building secure systems,” Twelfth International Conference on Computer Communications and Networks, ed. IEEE, IEEE Press, Oct. 2003.

[11] T. Ylonen, “The secure shell (SSH) transport layer protocol.” IETF RFC 4253, Jan. 2006.

Chanathip Namprempre Chanathip received the S.B. and M.Eng. in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology and the Ph.D. in Computer Science from the University of California, San Diego. Currently, she is an Assistant Professor in Electrical Engineering, Faculty of Engineering at Thammasat University, Thailand. Her research interests are cryptography and security.

Matthew N. Dailey Matthew received the B.S. and M.S. in Computer Science from North Carolina State University and the Ph.D. in Computer Science and Cognitive Science from the University of California, San Diego. He spent two years as a Research Scientist with Vision Robotics Corporation of San Diego, CA USA and two years as a Lecturer in the Computer Science and Information Technology programs at Sirindhorn International Institute of Technology, Thammasat University, Thailand. In 2006, he joined the Computer Science and Information Management department at the Asian Institute of Technology, Thailand, as an Assistant Professor. His research interests lie in machine learning, machine vision, robotics, and systems security.