GOTCHA Password Hackers![*]

Jeremiah Blocki
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
jblocki@cs.cmu.edu

Manuel Blum
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
mblum@cs.cmu.edu

Anupam Datta
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
danupam@cmu.edu

ABSTRACT

We introduce GOTCHAs (Generating panOptic Turing Tests to Tell Computers and Humans Apart) as a way of preventing automated offline dictionary attacks against user selected passwords. A GOTCHA is a randomized puzzle generation protocol, which involves interaction between a computer and a human. Informally, a GOTCHA should satisfy two key properties: (1) The puzzles are easy for the human to solve. (2) The puzzles are hard for a computer to solve even if it has the random bits used by the computer to generate the final puzzle — unlike a CAPTCHA [43]. Our main theorem demonstrates that GOTCHAs can be used to mitigate the threat of offline dictionary attacks against passwords by ensuring that a password cracker must receive constant feedback from a human being while mounting an attack. Finally, we provide a candidate construction of GOTCHAs based on Inkblot images. Our construction relies on the usability assumption that users can *recognize* the phrases that they originally used to describe each Inkblot image — a much weaker usability assumption than previous password systems based on Inkblots which required users to recall their phrase exactly. We conduct a user study to evaluate the usability of our GOTCHA construction. We also generate a GOTCHA challenge where we encourage artificial intelligence and security researchers to try to crack several passwords protected with our scheme.

Categories and Subject Descriptors

K.6.5 [**Computing Milieux**]: Security and Protection—*Authentication*

[*]This work was partially supported by the NSF Science and Technology TRUST and the AFOSR MURI on Science of Cybersecurity. The first author was also partially supported by an NSF Graduate Fellowship.

Keywords

Human Authentication; Passwords; GOTCHA; Inkblots; Offline Dictionary Attack; CAPTCHA; HOSP

1. INTRODUCTION

Any adversary who has obtained the cryptographic hash of a user's password can mount an automated brute-force attack to crack the password by comparing the cryptographic hash of the user's password with the cryptographic hashes of likely password guesses. This attack is called an offline dictionary attack, and there are many password crackers that an adversary could use [17]. Offline dictionary attacks against passwords are — unfortunately — powerful and commonplace [25]. Adversaries have been able to compromise servers at large companies (e.g., Zappos, LinkedIn, Sony, Gawker [5, 2, 9, 4, 1, 3]) resulting in the release of millions of cryptographic password hashes [1]. It has been repeatedly demonstrated that users tend to select easily guessable passwords [27, 18, 11], and password crackers are able to quickly break many of these passwords[39]. Offline attacks are becoming increasingly dangerous as computing hardware improves — a modern GPU can evaluate a cryptographic hash function like SHA2 about 250 million times per second [49] — and as more and more training data — leaked passwords from prior breaches — becomes available [25]. Symantec reported that compromised passwords have significant economic value to an adversary (e.g., compromised passwords are sold on black market for between $4 and $30) [22].

HOSPs (Human-Only Solvable Puzzles) were suggested by Canetti, Halevi and Steiner as a way of defending against offline dictionary attacks [14]. The basic idea is to change the authentication protocol so that human interaction is *required* to verify a password guess. The authentication protocol begins with the user entering his password. In response the server randomly generates a challenge — using the password as a source of randomness — for the user to solve. Finally, the server appends the user's response to the user's password, and verifies that the hash matches the record on the server. To crack the user's password offline the adversary must simultaneously guess the user's password and the answer to the corresponding puzzle. The challenge should be easy for a human to solve consistently so that a legitimate user can authenticate. To mitigate the threat of an offline dictionary attack the HOSP should be difficult for a

[1]In a few of these cases [3, 1] the passwords were stored in the clear.

computer to solve — even if it has all of the random bits used to generate the challenge.

The basic HOSP construction proposed by Canetti et al. [14] was to to fill a hard drive with regular CAPTCHAs (e.g., distorted text) by storing the puzzles without the answers. This solution only provides limited protection against an adversary because the number of unique puzzles that can be generated is bounded by the size of the hard drive (e.g., the adversary could pay people to solve all of the puzzles on the hard drive). See appendix B for more discussion. Finding a usable HOSP construction which does not rely on a very large dataset of pregenerated CAPTCHAs is an open problem. Several candidate HOSPs were experimentally tested [15] (they are called POSHs in the second paper), but the usability results were underwhelming.

Contributions.

We introduce a simple modification of HOSPs that we call GOTCHAs (Generating panOptic Turing Tests to Tell Computers and Humans Apart). We use the adjective Panoptic to refer to a world without privacy — there are no hidden random inputs to the puzzle generation protocol. The basic goal of GOTCHAs is similar to the goal of HOSPs — defending against offline dictionary attacks. GOTCHAs differ from HOSPs in two ways (1) Unlike a HOSP a GOTCHA may require human interaction during the *generation* of the challenge. (2) We relax the requirement that a user needs to be able to answer all challenges easily and consistently. If the user can remember his password during the authentication protocol then he will only ever see one challenge. We only require that the user must be able to answer this challenge consistently. If the user enters the wrong password during authentication then he may see new challenges. We do not require that the user must be able to solve these challenges consistently because authentication will fail in either case. We do require that it is difficult for a computer to distinguish between the "correct" challenge and an "incorrect" challenge. Our main theorem demonstrates that GOTCHAs like HOSPs can be used to defend against offline dictionary attacks. The goal of these relaxations is to enable the design of usable GOTCHAs.

We introduce a candidate GOTCHA construction based on Inkblot images. While the images are generated randomly by a computer, the human mind can easily imagine semantically meaningful objects in each image. To generate a challenge the computer first generates ten inkblot images (e.g., figure 1). The user then provides labels for each image (e.g., evil clown, big frog). During authentication the challenge is to match each inkblot image with the corresponding label. We empirically evaluate the usability of our inkblot matching GOTCHA construction by conducting a user study on Amazon's Mechanical Turk. Finally, we challenge the AI community to break our GOTCHA construction.

Organization.

The rest of the paper is organized as follows: We next discuss related work in section 1.1. We formally define GOTCHAs in section 2 and formalize the properties that a GOTCHA should satisfy. We present our candidate GOTCHA construction in section 3, and in section 3.1 we demonstrate how our GOTCHA could be integrated into an authentication protocol. We present the results from our user study in section 3.2, and in section 3.3 we challenge the AI and security communities to break our GOTCHA construction. In section 4 we prove that GOTCHAs like HOSPs can also be used to design a password storage system which mitigates the threat of offline attacks. We conclude by discussing future directions and challenges in section 5.

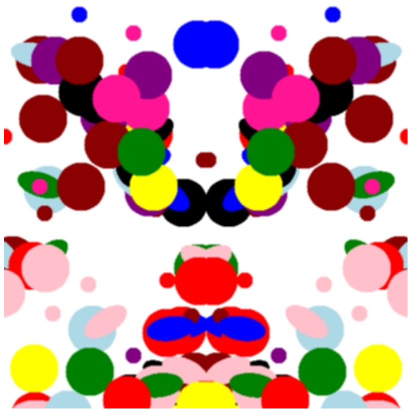
Figure 1: Randomly Generated Inkblot Image—An evil clown?

1.1 Related Work

Inkblots [42] have been proposed as an alternative way to generate and remember passwords. Stubblefield and Simon proposed showing the user ten randomly generated inkblot images, and having the user make up a word or a phrase to describe each image. These phrases were then used to build a 20 character password (e.g., users were instructed to take the first and last letter of each phrase). Usability results were moderately good, but users sometimes had trouble remembering their association. Because the Inkblots are publicly available there is also a security concern that Inkblot passwords could be guessable if different users consistently picked similar phrases to describe the same Inkblot.

We stress that our use of Inkblot images is different in two ways: (1) Usability: We do not require users to recall the word or phrase associated with each Inkblot. Instead we require user's to recognize the word or phrase associated with each Inkblot so that they can match each phrase with the appropriate Inkblot image. Recognition is widely accepted to be easier than the task of recall [7, 45]. (2) Security: We do not need to assume that it would be difficult for other humans to match the phrases with each Inkblot. We only assume that it is difficult for a computer to perform this matching automatically.

CAPTCHAs — formally introduced by Von Ahn et al. [43] — have gained widespread adoption on the internet to prevent bots from automatically registering for accounts. A CAPTCHA is a program that generates a puzzle — which should be easy for a human to solve and difficult for a computer to solve — as well as a solution. Many popular forms of CAPTCHAs (e.g., reCAPTCHA [44]) generate garbled text, which is easy [2] for a human to read, but difficult for a computer to decipher. Other versions of CAPTCHAs rely on the natural human capacity for audio [37] or image recognition [19].

[2]Admitedly some people would dispute the use of the label 'easy.'

CAPTCHAs have been used to defend against online password guessing attacks — users are sometimes required to solve a CAPTCHA before signing into their account. An alternative approach is to lock out a user after several incorrect guesses, but this can lead to denial of service attacks [16]. However, if the adversary has access to the cryptographic hash of the user's password, then he can circumvent all of these requirements and execute an automatic dictionary attack to crack the password offline. By contrast HOSPs — proposed by Canetti et al.[14] — were proposed to defend against offline attacks. HOSPs are in some ways similar to CAPTCHAs (Completely Automated Turing Tests to Tell Computers and Humans Apart) [43]. CAPTCHAs are widely used on the internet to fight spam by preventing bots from automatically registering for accounts. In this setting a CAPTCHA is sent to the user as a challenge, while the secret solution is used to grade the user's answer. The implicit assumption is that the answer and the random bits used to generate the puzzle remain hidden — otherwise a spam bot could simply regenerate the puzzle and the answer. While this assumption may be reasonable in the spam bot setting, it does not hold in our offline password attack setting in which the server has already been breached. A HOSP is different from a CAPTCHA in several key ways: (1) The challenge must remain difficult for a computer to solve even if the random bits used to generate the puzzle are made public. (2) There is no single correct answer to a HOSP. It is okay if different people give different responses to a challenge as long as people can respond to the challenges easily, and each user can consistently answer the challenges.

The only HOSP construction proposed in [14] involved stuffing a hard drive with unsolved CAPTCHAs. The problem of finding a HOSP construction that does not rely on a dataset of unsolved CAPTCHAs was left as an open problem [14]. Several other candidate HOSP constructions have been experimentally evaluated in subsequent work [15] (they are called POSHs in the second paper), but the usability results for every scheme that did not rely on a large dataset on unsolved CAPTCHAs were underwhelming.

GOTCHAs are very similar to HOSPs. The basic application — defending against offline dictionary attacks — is the same as are the key tools: exploiting the power of interaction during authentication, exploiting hard artificial intelligence problems. While the authentication with HOSPs is interactive, the initial generation of the puzzle is not. By contrast, our GOTCHA construction requires human interaction during the initial generation of the puzzle. This simple relaxation allows for the construction of new solutions. In the HOSP paper humans are simply modeled as a puzzle solving oracle, and the adversary is assumed to have a limited number of queries to a human oracle. We introduce a more intricate model of the human agent with the goal of designing more usable constructions.

Password Storage.

Password storage is an incredibly challenging problem. Adversaries have been able to compromise servers at many large companies (e.g., Zappos, LinkedIn, Sony, Gawker [5, 2, 9, 4, 1, 3]). For example, hackers were able to obtain 32 million plaintext passwords from RockYou using a simple SQL injection attack [1]. While it is considered an extremely poor security practice to store passwords in the clear [41], the practice is still fairly common [12, 3, 1]. Many other companies [4, 12] have used cryptographic hashes to store their passwords, but failed to adopt the practice of salting (e.g., instead of storing the cryptographic hash of the password $h(pw)$ the server stores $(h(pw, r), r)$ for a random string r [6]) to defend against rainbow table attacks. Rainbow tables, which consist of precomputed hashes, are often used by an adversary to significantly speed up a password cracking attack because the same table can be reused to attack each user when the passwords are unsalted [33].

Cryptographic hash functions like SHA1, SHA2 and MD5 — designed for fast hardware computation — are popular choices for password hashing. Unfortunately, this allows an adversary to try up to 250 million guesses per second on a modern GPU [49]. The BCRYPT [35] hash function was designed specifically with passwords in mind — BCRYPT was intentionally designed to be slow to compute (e.g., to limit the power of an adversary's offline attack). The BCRYPT hash function takes a parameter which allows the programmer to specify how costly the hash computation should be. The downside to this approach is that it also increases costs for the company that stores the passwords (e.g., if we want it to cost the adversary \$1,000 for every million guesses then it will also cost the company at least \$1,000 for every million login attempts).

Users are often advised (or required) to follow strict guidelines when selecting their password (e.g., use a mix of upper/lower case letters, include numbers and change the password frequently) [38]. However, empirical studies show that user's are are often frustrated by restricting policies and commonly forget their passwords [28, 29, 20] [3]. Furthermore, the cost of these restrictive policies can be quite high. For example, a Gartner case study [47] estimated that it cost over \$17 per password-reset call. Florencio and Herley [21] studied the economic factors that institutions consider before adopting password policies and found that they often value usability over security.

2. DEFINITIONS

In this section we seek to establish a theoretical basis for GOTCHAs. Several of the ideas behind our definitions are borrowed from theoretical definitions of CAPTCHAs [43] and HOSPs [14]. Like CAPTCHAs and HOSPs, GOTCHAs are based on the assumption that some AI problem is hard for a computer to solve, but easy for a person to solve. Ultimately, these assumptions are almost certainly false (e.g., because the human brain can solve a GOTCHA it is reasonable to believe that there exists a computer program to solve the problems). However, it may still be reasonable to assume that these problems cannot be solved by applying *known* ideas. By providing a formal definition of GOTCHAs we can determine whether or not a new *idea* can be used to break a candidate GOTCHA construction.

We use $c \in \mathcal{C}$ to denote the space of challenges that might be generated. We use $\mathcal{H}$ to denote the set of human users and $H(c, \sigma_t)$ to denote the response that a human $H \in \mathcal{H}$ gives to the challenge $c \in \mathcal{C}$ at time t. Here, σ_t denotes the state of the human's brain at time t. σ_t is supposed to encode our user's existing knowledge (e.g., vocabulary, experiences) as well as the user's mental state at time t (e.g., what is the user thinking about at time t). Be-

[3]In fact the resulting passwords are sometimes more vulnerable to an offline attack! [28, 29]

cause σ_t changes over time (e.g., new experiences) we use $H(c) = \{H(c, \sigma_t) \mid t \in \mathbb{N}\}$ to denote the *set* of all answers a human might give to a challenge c. We use $\mathcal{A}$ to denote the range of possible responses (answers) that a human might give to the challenges.

Definition 1. *Given a metric $d : \mathcal{A} \times \mathcal{A} \to \mathbb{R}$, we say that a human H can* consistently solve *a challenge $c \in \mathcal{C}$ with accuracy α if $\forall t \in \mathbb{N}$*

$$d(H(c, \sigma_0), H(c, \sigma_t)) \leq \alpha \ ,$$

where σ_0 denotes the state of the human's brain when he initially answers the challenge. If $|H(c)| = 1$ then we simply say that the human can consistently solve the challenge.

Notation: When we have a group of challenges $\langle c_1, \ldots, c_k \rangle$ we will sometimes write $H(\langle c_1, \ldots, c_k \rangle, \sigma_t) = \langle H(c_1, \sigma_t), \ldots, H(c_k, \sigma_t) \rangle$ for notational convenience. We use $y \sim \mathcal{D}$ to denote a random sample from the distribution $\mathcal{D}$, and we use $r \overset{\$}{\leftarrow} \{0,1\}^n$ to denote a element drawn from the set $\{0,1\}^n$ uniformly at random.

One of the requirements of a HOSP puzzle system [14] is that the human H must be able to *consistently* answer *any* challenge that is generated (e.g., $\forall c \in \mathcal{C}$, H can consistently solve c). These requirements seem to rule out promising ideas for HOSP constructions like Inkblots[15]. In this construction the challenge is a randomly generated inkblot image I, and the response $H(I, \sigma_0)$ is word or phrase describing what the user initially sees in the inkblot image (e.g., evil clown, soldier, big lady with a ponytail). User studies have shown that $H(I, \sigma_0)$ does not always match $H(I, \sigma_t)$ — the phrase describing what the user sees at time t [15]. In a few cases the errors may be correctable (e.g., capitalization, plural/singular form of a word), but oftentimes the phrase was completely different — especially if a long time passed in between trials [4]. By contrast, our GOTCHA construction does not require the user to remember the phrases associated with each Inkblot. Instead we rely on a much weaker assumption — the user can consistently recognize his solutions. We say that a human can recognize his solutions to a set of challenges if he can consistently solve a matching challenge (definition 2) in which he is asked to match each of his solutions with the corresponding challenge.

Definition 2. *Given an integer k, and a permutation $\pi : [k] \to [k]$, a* matching challenge *$\hat{c}_\pi = (\vec{c}, \vec{a}) \in \mathcal{C}$ of size k is given by a k-tuple of challenges $\vec{c} = \langle c_{\pi(1)}, \ldots, c_{\pi(k)} \rangle \in \mathcal{C}^k$ and solutions $\vec{a} = H(\langle c_1, \ldots, c_k \rangle, \sigma_0)$. The response to a matching challenge is a permutation $\pi' = H(\vec{c}_\pi, \sigma_t)$.*

For permutations $\pi : [k] \to [k]$ we use the distance metric

$$d_k(\pi_1, \pi_2) = |\{i \mid \pi_1(i) \neq \pi_2(i) \wedge 1 \leq i \leq k\}| \ .$$

$d_k(\pi_1, \pi_2)$ simply counts the number of entries where the permutations don't match. We say that a human can consistently recognize his solution to a matching challenge $\hat{c}_\pi$ with accuracy α if $\forall t . d_k(H(\hat{c}_\pi, \sigma_t), \pi) \leq \alpha$. We use $\{\pi' \mid d_k(\pi, \pi') \leq \alpha\}$ to denote the set of permutations π' that are α-close to π.

[4]We would add the requirement that the human must be able to consistently answer the challenges without spending time memorizing and rehearsing his response to the challenge. Otherwise we could just as easily force the user to remember a random string to append on to his password.

The puzzle generation process for a GOTCHA involves interaction between the human and a computer: (1) The computer generates a set of k challenges. (2) The human solves these challenges. (3) The computer uses the solutions to produce a final challenge [5]. Formally,

Definition 3. *A puzzle-system is a pair (G_1, G_2), where G_1 is a randomized challenge generator that takes as input 1^k (with k security parameter) and a pair of random bit strings $r_1, r_2 \in \{0,1\}^*$ and outputs k challenges $\langle c_1, \ldots, c_k \rangle \leftarrow G_1(1^k, r_1, r_2)$. G_2 is a randomized challenge generator that takes as input 1^k (security parameter), a random bit string $r_1 \in \{0,1\}^*$, and proposed answers $\vec{a} = \langle a_1, ..., a_k \rangle$ to the challenges $G_1(1^k, r_1, r_2)$ and outputs a challenge $\hat{c} \leftarrow G_2(1^k, r_1, \vec{a})$. We say that the puzzle-system is (α, β)-*usable *if*

$$\Pr_{H \overset{\$}{\leftarrow} \mathcal{H}} [\mathbf{Accurate}(H, \hat{c}, \alpha)] \geq \beta \ ,$$

whenever $\vec{a} = H(G_1(1^k, r_1, r_2), \sigma_0)$, where $\mathbf{Accurate}(H, \hat{c}, \alpha)$ denotes the event that the human H can consistently solve $\hat{c}$ with accuracy α.

In our authentication setting the random string r_1 is extracted from the user's password using a strong pseudorandom function **Extract**. To provide a concrete example of a puzzle-system, G_1 could be a program that generates a set of inkblot challenges $\langle I_1, \ldots, I_k \rangle$ using random bits r_1, selects a random permutation $\pi : [k] \to [k]$ using random bits r_2, and returns $\langle I_{\pi(1)}, \ldots, I_{\pi(k)} \rangle$. The human's response to an Inkblot — $H(I_j, \sigma_0)$ — is whatever he/she imagines when he sees the inkblot I_j for the first time (e.g., some people might imagine an evil clown when they look at figure 1). Finally, G_2 might generate Inkblots $\vec{c} = \langle I_1, \ldots, I_k \rangle$ using random bits r_1, and return the matching challenge $\hat{c}_\pi = (\vec{c}, \vec{a})$. In this case the matching challenge is for the user to match his labels with the appropriate Inkblot images to recover the permutation π. Observe that the final challenge — $\hat{c}_\pi$ — can only be generated after a round of *interaction* between the computer and a human. By contrast, the challenges in a HOSP must be generated automatically by a computer. Also notice that if G_2 is executed with a different random bit string r_1' then we do not require the resulting challenge to be consistently recognizable (e.g., if the user enters in the wrong password then authentication will fail regardless of how he solves the resulting challenge). For example, if the user enters the wrong password the user might be asked to match his labels $\langle \ell_{\pi(1)}, ..., \ell_{\pi(k)} \rangle = H(\langle I_{\pi(1)}, \ldots, I_{\pi(k)} \rangle, \sigma_0)$ with Inkblots $\langle I_1', \ldots, I_k' \rangle$ that he has never seen.

An adversary could attack a puzzle system by either (1) attempting to distinguish between the correct puzzle, and puzzles that might be meaningless to the human, or (2) by solving the matching challenge directly.

We say that an algorithm A can distinguish distributions $\mathcal{D}_1$ and $\mathcal{D}_2$ with advantage ϵ if

$$\left| \Pr_{x \sim \mathcal{D}_1} [A(x) = 1] - \Pr_{y \sim \mathcal{D}_2} [A(y) = 1] \right| \geq \epsilon \ .$$

Our formal definition of a GOTCHA is found in definition 4. Intuitively, definition 4 says that (1) The underlying

[5]We note that a HOSP puzzle system (G) [14] can be modeled as a GOTCHA puzzle system (G_1, G_2) where G_1 does nothing and G_2 simply runs G to generate the final challenge $\hat{c}$ directly.

puzzle-system should be usable — so that legitimate users can authenticate. (2) It should be difficult for the adversary to distinguish between the correct matching challenge (e.g., the one that the user will see when he types in the correct password), and an incorrect matching challenge (e.g., if the user enters the wrong password he will be asked to match his labels with different Inkblot images), and (3) It should be difficult for the adversary to distinguish between the user's matching, and a random matching drawn from a distribution R with sufficiently high minimum entropy.

DEFINITION 4. *A puzzle-system (G_1, G_2) is an $(\alpha, \beta, \epsilon, \delta, \mu)$-GOTCHA if (1) (G_1, G_2) is (α, β)-*usable *(2) Given a human $H \in \mathcal{H}$ no probabilistic polynomial time algorithm can distinguish between distributions*

$$\mathcal{D}_1 = \left\{ \begin{matrix} H(G_1(1^k, r_1, r_2), \sigma_0), \\ G_2(1^k, r_1, H(G_1(1^k, r_1, r_2), \sigma_0)) \end{matrix} \middle| r_1, r_2 \xleftarrow{\$} \{0,1\}^n \right\}$$

and

$$\mathcal{D}_2 = \left\{ \begin{matrix} H(G_1(1^k, r_1, r_2), \sigma_0), \\ G_2(1^k, r_3, H(G_1(1^k, r_1, r_2), \sigma_0)) \end{matrix} \middle| r_1, r_2, r_3 \xleftarrow{\$} \{0,1\}^n \right\}$$

with advantage greater than ϵ, and (3) Given a human $H \in \mathcal{H}$, there is a distribution $R(c)$ with $\mu(m)$ bits of minimum entropy such that no probabilistic polynomial time algorithm can distinguish between distributions

$$\mathcal{D}_3 = \left\{ \begin{matrix} H(G_1(1^k, r_1, r_2), \sigma_0) \\ G_2(1^k, r_1, H(G_1(1^k, r_1, r_2), \sigma_0)), \\ H(G_2(1^k, r_1, H(G_1(1^k, r_1, r_2), \sigma_0)), \sigma_0) \end{matrix} \middle| r_1, r_2 \xleftarrow{\$} \{0,1\}^n \right\}$$

and

$$\mathcal{D}_4 = \left\{ \begin{matrix} H(G_1(1^k, r_1, r_2), \sigma_0) \\ G_2(1^k, r_1, H(G_1(1^k, r_1, r_2), \sigma_0)), \\ R(G_2(1^m, r_1, \langle a_1, \ldots, a_m \rangle), \sigma_0) \end{matrix} \middle| r_1, r_2 \xleftarrow{\$} \{0,1\}^n \right\}$$

with advantage greater then δ.

2.1 Password Storage and Offline Attacks

To protect users in the event of a server breach organizations are advised to store salted password hashes — using a cryptographic hash function ($h : \{0,1\}^* \to \{0,1\}^n$) and a random bit string ($s \in \{0,1\}^*$) [38]. For example, if a user (u) chose the password (pw) the server would store the tuple $(u, s, h(s, pw))$. Any adversary who has obtained $(u, s, h(s, pw))$ (e.g., through a server breach) may mount a — fully automated — offline dictionary attack using powerful password crackers like John the Ripper [17]. To verify a guess pw' the adversary simply computes $h(s, pw')$ and checks to see if this hash matches $h(s, pw)$.

We assume that an adversary **Adv** who breaches the server can obtain the code for h, as well as the code for any GOTCHAs used in the authentication protocol. Given the code for h and the salt value s the adversary can construct a function

$$\mathbf{VerifyHash}(pw') = \begin{cases} 1 & \text{if } h(s, pw) = h(s, pw') \\ 0 & \text{otherwise.} \end{cases}$$

We also allow the adversary to have black box access to a GOTCHA solver (e.g., a human). We use c_H to denote the cost of querying a human and c_h to denote the cost of querying the function **VerifyHash**[6], and we use n_H (resp. n_h) to denote the number of queries to the human (resp. **VerifyHash**). Queries to the human GOTCHA solver are much more expensive than queries to the cryptographic hash function ($c_H \gg c_h$) [31]. For technical reasons we limit our analysis to conservative adversaries.

[6]The value of c_h may vary widely depending on the particular cryptographic hash function — it is inexpensive to evaluate SHA1, but BCRYPT [35] may be very expensive to evaluate.

DEFINITION 5. *We say that an adversary* **Adv** *is* conservative *if (1)* **Adv** *uses the cryptographic hash function h in a black box manner (e.g., the hash function h and the stored hash value are only used to construct a subroutine* **VerifyHash** *which is then used as a black box by* **Adv** *), (2) The pseudorandom function* **Extract** *is used as a black box, and (3) The adversary only queries a human about challenges generated using a password guess.*

It is reasonable to believe that our adversary is conservative. All existing password crackers (e.g., [17]) use the hash function as a black box, and it is difficult to imagine that the adversary would benefit by querying a human solver about Inkblots that are unrelated to the password.

We use $D \subseteq \{0,1\}^*$ to denote a dictionary of likely guesses that the adversary would like to try,

$$\mathbf{Cost}(\mathbf{Adv}, D) = (n_h c_h + n_H c_H)$$

to denote the cost of the queries that the adversary makes to check each guess in D, and $\mathbf{Succeed}(\mathbf{Adv}, D, pw)$ to denote the event that the adversary makes a query to **VerifyHash** that returns 1 (e.g., the adversary successfully finds the user's password pw). The adversary might use a computer program to try to solve some of the GOTCHAs — to save cost by not querying a human. However, in this case the adversary might fail to crack the password because the GOTCHA solver found the wrong solution to one of the challenges.

DEFINITION 6. *An adversary* **Adv** *is (C, γ, D)-successful if* $\mathbf{Cost}(\mathbf{Adv}, D) \leq C$, *and*

$$\Pr_{pw \xleftarrow{\$} D} \left[\mathbf{Succeed}(\mathbf{Adv}, D, pw) \right] \geq \gamma \ .$$

Our attack model is slightly different from the attack model in [14]. They assume that the adversary may ask a limited number of queries to a human challenge solution oracle. Instead we adopt an economic model similar to [10], and assume that the adversary is instead limited by a budget C, which may be used to either evaluate the cryptographic hash function h or query a human H.

3. INKBLOT CONSTRUCTION

Our candidate GOTCHA construction is based on Inkblots images. We use algorithm 1 to generate inkblot images. Algorithm 1 takes as input random bits r_1 and a security parameter k — which specifies the number of Inkblots to output. Algorithm 1 makes use of the randomized subroutine $\mathbf{DrawRandomEllipsePairs}(I, t, \mathit{width}, \mathit{height})$ which draws t pairs of ellipses on the image I with the specified width and height. The first ellipse in each pair is drawn at a random (x, y) coordinate on the left half of the image with a randomly selected color and angle α of rotation, and the second ellipse is mirrored on the right half of the image. Figure 1 is an example of an Inkblot image generated by algorithm 1.

Our candidate GOTCHA is given by the pair (G_1, G_2) — algorithms 2 and 3. G_1 runs algorithm 1 to generate k

Algorithm 1 GenerateInkblotImages

```
Input: Security Parameter $1^k$, Random bit string $r_1 \in \{0,1\}^*$.
for $j = 1, \ldots, k$ do
    $I_j \leftarrow$ new Blank Image    ▷ The following operations only use the random bit string $r_1$ as a source of randomness
    DrawRandomEllipsePairs $(I_j, 150, 60, 60)$
    DrawRandomEllipsePairs $(I_j, 70, 20, 20)$
    DrawRandomEllipsePairs $(I_j, 150, 60, 20)$
return $\langle I_1, \ldots, I_k \rangle$    ▷ Inkblot Images
```

Inkblot images, and then returns these images in permuted order — using a function **GenerateRandomPermutation** (k, r), which generates a random permutation $\pi : [k] \rightarrow [k]$ using random bits r. G_2 also runs algorithm 1 to generate k Inkblot images, and then outputs a matching challenge.

Algorithm 2 G_1

```
Input: Security Parameter $1^k$, Random bit strings $r_1, r_2 \in \{0,1\}^*$.
$\langle I_1, \ldots, I_k \rangle \leftarrow$ GenerateInkblotImages $(k, r_1)$
$\pi \leftarrow$ GenerateRandomPermutation $(k, r_2)$
return $\langle I_{\pi(1)}, \ldots, I_{\pi(k)} \rangle$
```

After the Inkblots $\langle I_{\pi(1)}, \ldots, I_{\pi(k)} \rangle$ have been generated, the human user is queried to provide labels $\ell_{\pi(1)}, \ldots, \ell_{\pi(k)}$ where

$$\langle \ell_{\pi(1)}, \ldots, \ell_{\pi(k)} \rangle = H\left(\langle I_{\pi(1)}, \ldots, I_{\pi(k)} \rangle, \sigma_0\right) \ .$$

In our authentication setting the server would store the labels $\ell_{\pi(1)}, \ldots, \ell_{\pi(k)}$ in permuted order. The final challenge — generated by algorithm 3 — is to match the Inkblot images $I_1, \ldots, I_k$ with the user generated labels $\ell_1, ..., \ell_k$ to recover the permutation π.

Algorithm 3 GenerateMatchingChallenge G_2

```
Input: Security Parameter $1^k$, Random bits $r_1 \in \{0,1\}^*$ and labels $\vec{a} = \langle \ell_{\pi(1)}, \ldots, \ell_{\pi(k)} \rangle$.
$\langle I_1, \ldots, I_k \rangle \leftarrow$ GenerateInkblotImages $(1^k, r_1)$
return $\hat{c}_\pi = (\vec{c}, \vec{a})$    ▷ Matching Challenge
```

Observation: Notice that if the random bits provided as input to **GenerateInkblotImages** and **GenerateMatchingChallenge** match that the user will see the same Inkblot images in the final matching challenge. However, if the random bits do not match (e.g., because the user typed the wrong password in our authentication protocol) then the user will see different Inkblot images. The labels $\ell_1, \ldots, \ell_k$ will be the same in both cases.

3.1 GOTCHA Authentication

To illustrate how our GOTCHAs can be used to defend against offline attacks we present the following authentication protocols: **Create Account** (protocol 3.1) and **Authenticate** (protocol 3.2). Communication in both protocols should take place over a secure channel. Both protocols involve several rounds of interaction between the user and the server. To create a new account the user sends his username/password to the server, the server responds by generating k Inkblot images $I_1, \ldots, I_k$, and the user provides a response $\langle \ell_1, \ldots, \ell_k \rangle = H(\langle I_1, \ldots, I_k \rangle, \sigma_0)$ based on his mental state at the time — the server stores these labels in permuted order $\ell_{\pi(1)}, \ldots, \ell_{\pi(k)}$ [7]. To authenticate later the user will have to match these labels with the corresponding inkblot images to recover the permutation π.

In section 4 we argue that the adversary who wishes to mount a cost effective offline attack needs to obtain constant feedback from a human. Following [14] we assume that the function **Extract** : $\{0,1\}^* \rightarrow \{0,1\}^n$ is a strong randomness extractor, which can be used to extract random strings from the user's password. Recall that $h : \{0,1\}^* \rightarrow \{0,1\}^*$ denotes a cryptographic hash function.

Protocol 3.1: Create Account

```
Security Parameters: $k$, $n$.
(User): Select username $(u)$ and password $(pw)$ and send $(u, pw)$ to the server.
(Server): Sends Inkblots $\langle I_1, \ldots, I_k \rangle$ to the user where:
    $r' \overset{\$}{\leftarrow} \{0,1\}^n$, $r_1 \leftarrow$ Extract $(pw, r')$, $r_2 \overset{\$}{\leftarrow} \{0,1\}^n$ and
    $\langle I_1, \ldots, I_k \rangle \leftarrow$ GenerateInkblotImages $(1^k, r_1)$
(User): Sends responses $\langle \ell_1, ..., \ell_k \rangle$ back to the server where:
    $\langle \ell_1, \ldots, \ell_k \rangle \leftarrow H(\langle I_1, \ldots, I_k \rangle, \sigma_0)$.
(Server): Store the tuple $t$ where $t$ is computed as follows:
Salt: $s \overset{\$}{\leftarrow} \{0,1\}^n$
$\pi \leftarrow$ GenerateRandomPermutation $(k, r_2)$.
$h_{pw} \leftarrow h(u, s, pw, \pi(1), ..., \pi(k))$
$t \leftarrow (u, r', s, h_{pw}, \ell_{\pi(1)}, \ldots, \ell_{\pi(k)})$
```

Protocol 3.2: Authenticate

```
Security Parameters: $k$, $n$.
Usability Parameter: $\alpha$
(User): Send username $(u)$ and password $(pw')$ — $pw'$ may or may not be correct.
(Server): Sends challenge $\hat{c}$ to the user where $\hat{c}$ is computed as follows:
    Find $t = (u, r', s, h_{pw}, \ell_{\pi(1)}, \ldots, \ell_{\pi(k)})$
    $r'_1 \leftarrow$ Extract $(pw', r')$
    $\langle I'_1, ..., I'_k \rangle \leftarrow$ GenerateInkblotImages $(r'_1, k)$
    $\hat{c}_\pi \leftarrow (\langle I_1, ..., I_k \rangle, \langle \ell_{\pi(1)}, \ldots, \ell_{\pi(k)} \rangle)$
(User): Solves $\hat{c}_\pi$ and sends the answer $\pi' = H(\hat{c}, \sigma_t)$.
(Server):
for all $\pi_0$ s.t $d_k(\pi_0, \pi') \le \alpha$ do
    $h_{pw,0} \leftarrow h(u, s, pw', \pi_0(1), ..., \pi_0(k))$
    if $h_{pw,0} = h_{pw}$ then
        Authenticate
Deny
```

Our protocol could be updated to allow the user to reject challenges he found confusing during account creation

[7]For a general GOTCHA, protocol 3.1 would need to have an extra round of communication. The server would send the user the final challenge generated by G_2 and the user would respond with $H(G_2(,), \sigma_0)$. Protocol 3.1 takes advantage of the fact that $\pi = H(G_2(,), \sigma_0)$ is already known.

in protocol 3.1. In this case the server would simply note that the first GOTCHA was confusing and generate a new GOTCHA. Once our user has created an account he can login by following protocol 3.2.

Claim 1 says that a legitimate user can successfully authenticate if our Inkblot construction satisfies the usability requirements of a GOTCHA. The proof of claim 1 can be found in appendix A.

CLAIM 1. *If (G_1, G_2) is a $(\alpha, \beta, \epsilon, \delta, \mu)$-GOTCHA then at least β-fraction of humans can successfully authenticate using protocol 3.2 after creating an account using protocol 3.1.*

One way to improve usability of our authentication protocol is to increase the neighborhood of acceptably close matchings by increasing α. The disadvantage is that the running time for the server in protocol 3.2 increases with the size of α. Claim 2 bounds the time needed to enumerate over all close permuations. The proof of claim 2 can be found in appendix A.

CLAIM 2. *For all permutations $\pi : [k] \rightarrow [k]$ and $\alpha \geq 0$*

$$\left|\left\{\pi' \mid d_k\left(\pi, \pi'\right) \leq \alpha\right\}\right| \leq 1 + \sum_{i=2}^{\alpha} \binom{k}{i} i! \ .$$

For example, if the user matches $k = 10$ Inkblots and we want to accept matchings that are off by at most $\alpha = 5$ entries then the server would need to enumerate over at most $36,091$ permutations[8]. Organizations are already advised to use password hash functions like BCRYPT [35] which intentionally designed to be slower than standard cryptographic hash functions — often by a factor of millions. Instead of making the hash function a million times slower to evaluate the server might instead make the hash function a thousand times slower to evaluate and use these extra computation cycles to enumerate over close permutations. The organization's trade-off is between: security, usability and the resources that it needs to invest during the authentication process.

We observe that an adversary mounting an online attack would be naturally rate limited because he would need to solve a GOTCHA for each new guess. Protocol 3.2 could also be supplemented with a k-strikes policy — in which a user is locked out for several hours after k incorrect login attempts — if desired.

3.2 User Study

To test our candidate GOTCHA construction we conducted an online user study[9]. We recruited participants through Amazon's Mechanical Turk to participate in our study. The study was conducted in two phases. In phase 1 we generated ten random Inkblot images for each participant, and asked each participant to provide labels for their Inkblot images. Participants were advised to use creative titles (e.g., evil clown, frog, lady with poofy dress) because they would not need to remember the exact titles that they used. Participants were paid \$1 for completing this first phase. A total of 70 users completed phase 1.

After our participants completed the first phase we waited ten days before asking our participants to return and complete phase 2. During phase 2 we showed each participant the Inkblot images they saw in phase 1 (in a random order) as well as the titles that they created during phase 1 (in alphabetical order). Participants were asked to match the labels with the appropriate image. The purpose of the longer waiting time was to make sure that participants had time to forget their images and their labels. Participants were paid an additional \$1 for completing phase 2 of the user study. At the beginning of the user study we let participants know that they would be paid during phase 2 even if their answers were not correct. We adopted this policy to discourage cheating (e.g., using screen captures from phase 1 to match the images and the labels) and avoid positively biasing our results.

We measured the time it took each participant to complete phase 1. Our results are summarized in table 1. It is quite likely that some participants left their computer in the middle of the study and returned later to complete the study (e.g., one user took 57.5 minutes to complete the study). While we could not measure time away from the computer, we believe that it is likely that at least 9 of our participants left the computer. Restricting our attention to the other 61 participants who took at most 20 minutes we get an adjusted average completion time of 6.2 minutes.

Fifty-eight of our participants returned to complete phase 2 by taking our matching test. It took these participants 4.5 minutes on average to complete the matching test. Seventeen of our participants correctly matched all ten of their labels, and 69% of participants matched at least 5 out of ten labels correctly. Our results are summarized in table 2.

	Phase 1	Phase 2
Average	9.3	4.5
StdDev	9.6	3
Max	57.5	18.5
Min	1.4	1.6
Average $\leq$ 20	6.2	N/A

Table 1: Completion Times

α-accurate	# participants	$\frac{\text{\# participants}}{58}$	$\frac{\left\|\left\{\pi' \mid d_{10}\left(\pi,\pi'\right)\leq\alpha\right\}\right\|}{10!}$
$\alpha = 0$	17	0.29	2.76×10^{-7}
$\alpha = 2$	22	0.38	1.27×10^{-5}
$\alpha = 3$	26	0.45	7.88×10^{-5}
$\alpha = 4$	34	0.59	6.00×10^{-4}
$\alpha = 5$	40	0.69	3.66×10^{-3}

Table 2: Usability Results: Fraction of Participants who would have authenticated with accuracy parameter α

Discussion.

Our user study provides evidence that our construction is at least $(0, 0.29)$-usable or $(5, 0.69)$-usable. While this means that our Inkblot Matching GOTCHA could be used by a significant fraction of the population to protect their pass-

[8]A more precise calculation reveals that there are exactly $13,264$ permutations s.t. $d_{10}\left(\pi', \pi\right) \leq 5$ and a random permuation π' would only be accepted with probability 3.66×10^{-3}

[9]Our study protocol was approved for exemption by the Institutional Review Board (IRB) at Carnegie Mellon University (IRB Protocol Number: HS13-219).

words during authentication it also means that the use of our GOTCHA would have to be voluntary so that users who have difficulty won't get locked out of their accounts. Another approach would be to construct different GOTCHAs and allow users to choose which GOTCHA to use during authentication.

Study Incentives: There is evidence that the lack of monetary incentives to perform well on our matching test may have negatively influenced the results (e.g., some participants may have rushed through phase 1 of the study because their payment in round 2 was independent of their ability to match their labels correctly). For example, none of our 18 fastest participants during phase 1 matched all of their labels correctly, and — excluding participants we believe left their computer during phase 1 (e.g., took longer than 20 minutes) — on average participants who failed to match at least five labels correctly took 2 minutes less time to complete phase 1 than participants who did.

Time: We imagine that some web services may be reluctant to adopt GOTCHAs out of fear driving away customers who don't want to spend time labeling Inkblot images [21]. However, we believe that for many high security applications (e.g., online banking) the extra security benefits of GOTCHAs will outweigh the costs — GOTCHAs might even help a bank keep its customers by providing extra assurance that users' passwords are secure. We are looking at modifying our Inkblot generation algorithm to produce Inkblots which require less "mental effort" to label. In particular could techniques like Perlin Noise [34] be used to generate Inkblots that can be labeled more quickly and matched more accurately?

Accuracy: We believe that the usability of our Inkblot Matching GOTCHA construction can still be improved. One simple way to improve the usability of our GOTCHA construction would be to allow the user to reject Inkblot images that were confusing. We also believe that usability could be improved by providing users with specific strategies for creating their labels (e.g., we found that simple labels like "a voodoo mask" were often mismatched, while more elaborate stories like "A happy guy on the ground, protecting himself from ticklers" were rarely mismatched).

3.3 An Open Challenge to the AI Community

We envision a rich interaction between the security community and the artificial intelligence community. To facilitate this interaction we present an open challenge to break our GOTCHA scheme.

Challenge Setup.

We chose several random passwords $(pw_1, ..., pw_4) \overset{\$}{\leftarrow} \{0, 10^7\}$ and $pw_5 \overset{\$}{\leftarrow} \{0, 10^8\}$. We used a function **GenerateInkblots** $(pw_i, 10)$ to generate ten inkblots $I_1^i, ..., I_{10}^i$ for each password, and we had a human label each inkblot image $\langle \ell_1^i, \ldots, \ell_{10}^i \rangle \leftarrow H\left(\langle I_1^i, \ldots, I_{10}^i \rangle, \sigma_0\right)$. We selected a random permutation $\pi_i : [10] \to [10]$ for each account, and generated the tuple

$$T_i = \left(s_i, h\left(pw_i, s_i, \pi_i(1), ..., \pi_i(10)\right), \ell^i_{\pi_i(1)}, ..., \ell^i_{\pi_i(10)}\right) ,$$

where s_i is a randomly selected salt value and h is a cryptographic hash function. We are releasing the source code that we used to generate the Inkblots and evaluate the hash function h along with the tuples $T_1, ..., T_5$ — see `http://www.cs.cmu.edu/~jblocki/GOTCHA-Challenge.html`.
Challenge: Recover each password pw_i.

Approaches.

One way to accomplish this goal would be to enumerate over every possible password guess pw_i' and evaluate $h\left(pw_i', s_i, \pi(1), ..., \pi(10)\right)$ for every possible permutation $\pi : [10] \to [10]$. However, the goal of this challenge is to see if AI techniques can be applied to attack our GOTCHA construction. We intentionally selected our passwords from a smaller space to make the challenge more tractable for AI based attacks, but to discourage participants from trying to brute force over all password/permutation pairs we used BCRYPT (Level 15)[10] — an expensive hash function — to encrypt the passwords. Our implementation allows the Inkblot images to be generated very quickly from a password guess pw' so an AI program that can use the labels in the password file to distinguish between the correct Inkblots returned by **GenerateInkblots** $(pw_i, 10)$ and incorrect Inkblots returned by **GenerateInkblots** $(pw_i', 10)$ would be able to quickly dismiss incorrect guesses. Similarly, an AI program which generates a small set of likely permutations for each password guess could allow an attacker to quickly dismiss incorrect guesses.

4. ANALYSIS: COST OF OFFLINE ATTACKS

In this section we argue that our password scheme (protocols 3.2 and 3.1) significantly mitigates the threat of offline attacks. An informal interpretation of our main technical result — Theorem 1 — is that either (1) the adversary's offline attack is prohibitively expensive (2) there is a good chance that adversary's offline attack will fail, or (3) the underlying GOTCHA construction can be broken. Observe that the security guarantees are still meaningful even if the security parameters ϵ and δ are not negligably small.

THEOREM 1. *Suppose that our user selects his password uniformly at random from a set D (e.g., $pw \overset{\$}{\leftarrow} D$) and creates his account using protocol 3.1. If algorithms 2 and 3 are an (ϵ, δ, μ)-GOTCHA then no conservative offline adversary is $\left(C, \gamma + \epsilon + \delta + \frac{n_H}{|D|}, D\right)$-successful for $C < \gamma |D| 2^{\mu(k)} c_h + n_H c_H$*

Proof of Theorem 1. (Sketch) We use a hybrid argument. An adversary who breaches the server is able to recover the tuple $t = \left(u, r', s, h\left(u, s, pw, \pi(1), \ldots, \pi(k)\right), \ell_{\pi(1)}, \ldots, \ell_{\pi(k)}\right)$ as well as the code for the cryptographic hash function h and the code for our GOTCHA — (G_1, G_2).

1. World 0: W_0 denotes the real world in which the adversary has recovered the tuple

   $$t_0 = \left(u, r', s, h\left(u, s, pw, \pi(1), \ldots, \pi(k)\right), \ell_{\pi(1)}, \ldots, \ell_{\pi(k)}\right)$$

   as well as the code for the cryptographic hash function h and the code for our GOTCHA — (G_1, G_2). Because the adversary **Adv** is conservative it constructs the function

[10]The level parameter specifies the computation complexity of hashing. The amount of work necessary to evaluate the BCRYPT hash function increases exponentially with the level so in our case the work increases by a factor of 2^{15}.

$$\mathbf{VerifyHash}\left(pw', \pi'\right) = \begin{cases} 1 & \text{if } pw' = pw \text{ and } \pi' = \pi \\ 0 & \text{otherwise.} \end{cases},$$

and uses **VerifyHash** as a blackbox. We say that **Adv** queries a human H about password pw' if it queries H for $H\left(\mathbf{GenerateInkblotImages}\left(1^k, \mathbf{Extract}\left(pw', r'\right)\right)\right)$, and we let $D' \subseteq D$ denote the set of passwords for which the adversary queries a human.

2. World 1: W_1 denotes a hypothetical world that is similar to W_0 except that **VerifyHash** function the adversary uses as a blackbox is replaced with the following incorrect version

$$\mathbf{VerifyHash}^1\left(pw', \pi'\right) = \begin{cases} 1 & \text{if } pw' \notin D', pw' = pw \text{ and } \pi' = \pi \\ 0 & \text{otherwise.} \end{cases},$$

where $D' \subseteq D$ is a subset of passwords which denotes the set of passwords for which the adversary makes queries to a human in the real world.

3. World 2: W_2 denotes a hypothetical world that is similar to W_1 except that **VerifyHash**^{1} function the adversary uses as a blackbox is replaced with the following incorrect version

$$\mathbf{VerifyHash}^2\left(pw', \pi'\right) = \begin{cases} 1 & \text{if } \pi' = R\left(G_2\left(1^k, \mathbf{Extract}\left(pw', r'\right), \ell_1, \ldots, \ell_k\right)\right) \\ & \text{and } pw' \notin D', pw' = pw \\ 0 & \text{otherwise.} \end{cases},$$

where R is a distribution with minimum entropy $\mu(k)$ as in definition 4.

4. World 3: W_3 denotes a hypothetical real world which is similar to world 2, except that the labels $\ell_{\pi(1)}, \ldots, \ell_{\pi(k)}$ are replaced with the labels $\ell'_{\pi'(1)}, \ldots, \ell'_{\pi'(k)}$, where $\pi' : [k] \rightarrow [k]$ is a new random permutation, and the labels ℓ'_i are for a completely unrelated set of Inkblot challenges

$$\ell'_1, \ldots, \ell_k \leftarrow H\left(G_1\left(1^k, x_1, x_2\right)\right),$$

where $x_1, x_2 \in \{0,1\}^n$ are freshly chosen random value.

In world 3 it is easy to bound the adversary's probability of success. No adversary is (C, γ, D)-successful for $C < \gamma|D|2^{\mu(k)}c_h$, because the fake Inkblot labels are not correlated with the actual Inblots that were generated with the real password. Our particular advesary cannot be (C, γ, D)-successful for $C < \gamma|D|2^{\mu(k)}c_h + |D'|c_H$. In world 2 the adversary might improve his chances of success by looking at the Inblot labels, but by definition of $(\alpha, \beta, \epsilon, \delta, \mu)$-GOTCHA his chances change by at most δ. In world 1 the adversary might further improve his chances of success, but by definition of $(\alpha, \beta, \epsilon, \delta, \mu)$-GOTCHA his chances improve by at most ϵ. Finally, in world 0 the adversary improves his chances by at most $|D'|/|D|$ by querying the human about passwords in D'. □

5. DISCUSSION

We conclude by discussing some key directions for future work.

Other GOTCHA Constructions.

Because GOTCHAs allow for human feedback during puzzle generation — unlike HOSPs [14] — our definition potentially opens up a much wider space of potential GOTCHA constructions. One idea might be to have a user rate/rank random items (e.g., movies, activities, foods). By allowing human feedback we could allow the user to dismiss potentially confusing items (e.g., movies he hasn't seen, foods about which he has no strong opinion). There is some evidence that this approach could provide security (e.g., Narayanan and Shmatikov showed that a Netflix user can often be uniquely identified from a few movie ratings [32].).

Obfuscating CAPTCHAs.

If it were possible to efficiently obfuscate programs then it would be easy to construct GOTCHAs from CAPTCHAs (e.g., just obfuscate a program that returns the CAPTCHA without the answer). Unfortunately, there is no general program obsfuscator [8]. However, the approach may not be entirely hopeless. Point functions [46] can be obfuscated, and our application is similar to a point function — the puzzle generator G_2 in an GOTCHA only needs to generate a human solvable puzzle for one input. Recently, multilinear maps have been used to obfuscate conjunctions [13] and to obfuscate NC^1 circuits [23] [11]. Could similar techniques be used obfuscate CAPTCHAs?

Exploiting The Power of Interaction.

Can interaction be exploited and used to improve security or usability in human-authentication? While interaction is an incredibly powerful tool in computer security (e.g., nonces [36], zero-knowledge proofs [24], secure multiparty computation [48]) and in complexity theory[12], human authentication typically does not exploit interaction with the human (e.g., the user simply enters his password). We view the idea behind HOSPs and GOTCHAs — exploiting interaction to mitigate the threat of offline attacks — as a positive step in this direction. Could interaction be exploited to reduce memory burden on the user by allowing a user to reuse the same secret to authenticate to multiple different servers? The human-authentication protocol of Hopper, et al. [26] — based on the noisy parity problem — could be used by a human to repeatedly authenticate over an insecure channel. Unfortunately, the protocol is slow and tedious for a human to execute, and it can be broken if the adversary is able to ask adaptive parity queries [30].

6. REFERENCES

[1] Rockyou hack: From bad to worse. http://techcrunch.com/2009/12/14/rockyou-hack-

[11] The later result used a weaker notion of obfuscation known as "indistinguishability obfuscation," which (loosely) only guarantees that the adversary cannot distinguish between the obfuscations of two circuits which compute the same function.

[12] A polynomial time verifier can verify **PSPACE**-complete languages by interacting with a powerful prover [40], by contrast the same verifier can only check proofs of **NP**-Complete languages without interaction.

security-myspace-facebook-passwords/, December 2009. Retrieved 9/27/2012.

[2] Update on playstation network/qriocity services. http://blog.us.playstation.com/2011/04/22/update-on-playstation-network-qriocity-services/, April 2011. Retrieved 5/22/2012.

[3] Data breach at ieee.org: 100k plaintext passwords. http://ieeelog.com/, September 2012. Retrieved 9/27/2012.

[4] An update on linkedin member passwords compromised. http://blog.linkedin.com/2012/06/06/linkedin-member-passwords-compromised/, June 2012. Retrieved 9/27/2012.

[5] Zappos customer accounts breached. http://www.usatoday.com/tech/news/story/2012-01-16/mark-smith-zappos-breach-tips/52593484/1, January 2012. Retrieved 5/22/2012.

[6] S. Alexander. Password protection for modern operating systems. *;login*, June 2004.

[7] A. Baddeley. *Human memory: Theory and practice.* Psychology Pr, 1997.

[8] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im) possibility of obfuscating programs. In *Advances in Cryptology-CRYPTO 2001*, pages 1–18. Springer, 2001.

[9] S. Biddle. Anonymous leaks 90,000 military email accounts in latest antisec attack. http://gizmodo.com/5820049/anonymous-leaks-90000-military-email-accounts-in-latest-antisec-attack, July 2011. Retrieved 8/16/2011.

[10] J. Blocki, M. Blum, and A. Datta. Naturally rehearsing passwords. In *Advances in Cryptology-ASIACRYPT 2013 (to appear)*.

[11] J. Bonneau. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *Proc. of Oakland*, pages 538–552, 2012.

[12] J. Bonneau and S. Preibusch. The password thicket: technical and market failures in human authentication on the web. In *Proc. of WEIS*, volume 2010, 2010.

[13] Z. Brakerski and G. N. Rothblum. Obfuscating conjunctions. In *Advances in Cryptology-CRYPTO 2013*, pages 416–434. Springer, 2013.

[14] R. Canetti, S. Halevi, and M. Steiner. Mitigating dictionary attacks on password-protected local storage. In *Advances in Cryptology-CRYPTO 2006*, pages 160–179. Springer, 2006.

[15] W. Daher and R. Canetti. Posh: A generalized captcha with security applications. In *Proceedings of the 1st ACM workshop on Workshop on AISec*, pages 1–10. ACM, 2008.

[16] M. Dailey and C. Namprempre. A text graphics character captcha for password authentication. In *TENCON 2004. 2004 IEEE Region 10 Conference*, pages 45–48. IEEE, 2004.

[17] S. Designer. John the Ripper. http://www.openwall.com/john/, 1996-2010.

[18] K. Doel. Scary logins: Worst passwords of 2012 and how to fix them. http://www.prweb.com/releases/2012/10/prweb10046001.htm, 2012. Retrieved 1/21/2013.

[19] J. Elson, J. R. Douceur, J. Howell, and J. Saul. Asirra: a captcha that exploits interest-aligned manual image categorization. In *Proc. of CCS*.

[20] D. Florencio and C. Herley. A large-scale study of web password habits. In *Proceedings of the 16th international conference on World Wide Web*, pages 657–666. ACM, 2007.

[21] D. Florêncio and C. Herley. Where do security policies come from? In *Proceedings of the Sixth Symposium on Usable Privacy and Security*, pages 1–14. ACM, 2010.

[22] M. Fossi, E. Johnson, D. Turner, T. Mack, J. Blackbird, D. McKinney, M. K. Low, T. Adams, M. P. Laucht, and J. Gough. Symantec report on the undergorund economy, November 2008. Retrieved 1/8/2013.

[23] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *Proc. of FOCS (to appear)*, 2013.

[24] O. Goldreich, A. Sahai, and S. Vadhan. Can statistical zero knowledge be made non-interactive? or on the relationship of SZK and NISZK. In *Advances in Cryptology-CRYPTO 1999*, pages 467–484, 1999.

[25] D. Goodin. Why passwords have never been weaker and crackers have never been stronger. http://arstechnica.com/security/2012/08/passwords-under-assault/, 2012.

[26] N. J. Hopper and M. Blum. Secure human identification protocols. In *Advances in Cryptology-ASIACRYPT 2001*, pages 52–66. Springer, 2001.

[27] Imperva. Consumer password worst practices. 2010. Retrived 1/22/2013.

[28] S. Komanduri, R. Shay, P. Kelley, M. Mazurek, L. Bauer, N. Christin, L. Cranor, and S. Egelman. Of passwords and people: measuring the effect of password-composition policies. In *Proc. of CHI*, pages 2595–2604, 2011.

[29] H. Kruger, T. Steyn, B. Medlin, and L. Drevin. An empirical assessment of factors impeding effective password management. *Journal of Information Privacy and Security*, 4(4):45–59, 2008.

[30] E. Kushilevitz and Y. Mansour. Learning decision trees using the Fourier spectrum. *SIAM J. Comput.*, 22(6):1331–1348, 1993.

[31] M. Motoyama, K. Levchenko, C. Kanich, D. McCoy, G. M. Voelker, and S. Savage. Re: Captchas–understanding captcha-solving services in an economic context. In *USENIX Security Symposium*, volume 10, 2010.

[32] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *Proc. of the 2008 IEEE Symposium on Security and Privacy*, pages 111–125. IEEE, 2008.

[33] P. Oechslin. Making a faster cryptanalytic time-memory trade-off. *Advances in Cryptology-CRYPTO 2003*, pages 617–630, 2003.

[34] K. Perlin. Implementing improved perlin noise. *GPU Gems*, pages 73–85, 2004.

[35] N. Provos and D. Mazieres. Bcrypt algorithm.

[36] P. Rogaway. Nonce-based symmetric encryption. In *Fast Software Encryption*, pages 348–358. Springer, 2004.

[37] G. Sauer, H. Hochheiser, J. Feng, and J. Lazar. Towards a universally usable captcha. In *Proceedings of the 4th Symposium on Usable Privacy and Security*, 2008.

[38] K. Scarfone and M. Souppaya. NIST special publication 800-118: Guide to enterprise password management (draft), 2009.

[39] D. Seeley. Password cracking: A game of wits. *Communications of the ACM*, 32(6):700–703, 1989.

[40] A. Shamir. Ip= pspace. *Journal of the ACM (JACM)*, 39(4):869–877, 1992.

[41] A. Singer. No plaintext passwords. *;login: THE MAGAZINE OF USENIX & SAGE*, 26(7), November 2001. Retrieved 8/16/2011.

[42] A. Stubblefield and D. Simon. Inkblot authentication. Technical report, 2004.

[43] L. Von Ahn, M. Blum, N. Hopper, and J. Langford. Captcha: Using hard ai problems for security. *Advances in Cryptology-EUROCRYPT 2003*, pages 646–646, 2003.

[44] L. Von Ahn, B. Maurer, C. McMillen, D. Abraham, and M. Blum. recaptcha: Human-based character recognition via web security measures. *Science*, 321(5895):1465–1468, 2008.

[45] M. J. Watkins and J. M. Gardiner. An appreciation of generate-recognize theory of recall. *Journal of Verbal Learning and Verbal Behavior*, 18(6):687–704, 1979.

[46] H. Wee. On obfuscating point functions. In *Proc. of STOC*, pages 523–532. ACM, 2005.

[47] R. Witty, K. Brittain, and A. Allen. Justify identity management investment with metrics. Gartner Group report, 2004.

[48] A. C. Yao. Protocols for secure computations. In *Proc. of FOCS*, pages 160–164, 1982.

[49] A. Zonenberg. Distributed hash cracker: A cross-platform gpu-accelerated password recovery system. *Rensselaer Polytechnic Institute*, page 27, 2009.

APPENDIX

A. MISSING PROOFS

Reminder of Claim 1. *If (G_1, G_2) is a $(\alpha,\beta,\epsilon,\delta,\mu)$-GOTCHA then at least β-fraction of humans can sucessfully authenticate using protocol 3.2 after creating an account using protocol 3.1.*

Proof of Claim 1. A legitimate user $H \in \mathcal{H}$ will use the same passwords in protocols 3.1 and 3.2. Hence,

$$r_1' = \textbf{Extract}\left(pw', r'\right) = \textbf{Extract}\left(pw, r'\right) = r_1 \ ,$$

and the final matching challenge $\hat{c}_\pi$ is the same one that would be generated by $G_2\left(1^k, r_1, H\left(G_1\left(1^k, r_1, r_2\right), \sigma_0\right)\right)$. If $\hat{c}_\pi$ is consistently solvable with accuracy α by H — by definition 4 this is the case for at least β-fraction of users — then it follows that

$$d_k\left(\pi, \pi', \sigma_t\right) \le \alpha \ ,$$

where $H\left(G_1\left(1^k, r_1, r_2\right)\right)$. For some π_0 (namely $\pi_0 = \pi$) s.t. $d_k\left(\pi_0, \pi'\right) \le \alpha$ it must be the case that

$$\begin{aligned} h_{pw,0} &= h\left(u, s, pw', \pi_0(1), ..., \pi_0(k)\right) \\ &= h\left(u, s, pw, \pi(1), ..., \pi(k)\right) \\ &= h_p w \ , \end{aligned}$$

and protocol 3.2 accepts. □

Reminder of Claim Claim 2. *For all permutations $\pi : [k] \to [k]$ and $\alpha \ge 0$*

$$\left|\left\{\pi' \mid d_k\left(\pi, \pi'\right) \le \alpha\right\}\right| \le 1 + \sum_{i=2}^{\alpha} \binom{k}{i} i! \ .$$

Proof of 2. It suffices to show that $\binom{k}{j} j! \ge \left|\left\{\pi' \mid d_k\left(\pi, \pi'\right) = j\right\}\right|$. We first choose the j unique indices $i_1, \ldots, i_j$ on which π and π' differ — there are $\binom{k}{j}$ ways to do this. Once we have fixed our indices $i_1, \ldots, i_j$ we define $\pi'(k) = \pi(k)$ for each $k \notin \{i_1, \ldots, i_j\}$. Now $j!$ upperbounds the number of ways of selecting the remaining values $\pi'(i_k)$ s.t. $\pi(i_k) \ne \pi'(i_k)$ for all $k \le j$. □

B. HOSP: PRE-GENERATED CAPTCHAS

The HOSP construction proposed by [14] was to simply fill several high capacity hard drives with randomly generated CAPTCHAs — discarding the solutions. Once we have compiled a database large D of CAPTCHAs we can use algorithm 4 as our challenge generator — simply return a random CAPTCHA from D. The advantage of this approach is that we can make use of already tested CAPTCHA solutions so there is no need to make hardness assumptions about new AI problems. The primary disadvantage of this approach is that the size of the database D will be limited by economic considerations — storage isn't free. While $|D|$ the number of CAPTCHAs that could be stored on a hard drive may be large, it is not exponentially large. An adversary could theoretically pay humans to solve every puzzle in D at which point the scheme would be completely broken.

Economic Cost.

Suppose that two 4 TB hard drives are filled will text CAPTCHAS [13]. Let S be the space required to store one

[13]At the time of submission a 4 TB hard drive can be purchased on Amazon for less than $162.

Algorithm 4 GenerateChallenge

```
Input: Random bits $r \in \{0,1\}^n$, Database $D = \{P_1, ..., P_{2^n}\}$ of CAPTCHAs
return $P_r$
```

CAPTCHA, and let C_H denote the cost of paying a human to solve a CAPTCHA. We use the values $S = 8$ KB [14] and $C_H = \$0.001$ [15]. In this case $|D| = \frac{4\ TB}{8KB} \approx 10^9$ so we can store a billion unsolved CAPTCHAs on the hard drives. It would cost the adversary $|D|C_H = \$1,000,000$ to solve all of the CAPTCHAs — or $\$500,000$ to solve half of them. The up front cost of this attack may be large, but once the adversary has solved the CAPTCHAs he can execute offline dictionary attacks against every user who had an account on the server. Many server breaches have resulted in the release of password records for millions of accounts [5, 4, 2, 1]. If each cracked password is worth between \$4 and \$30 [22] then it may be easily worth the cost to pay humans to solve every CAPTCHA in D.

[14] The exact value of S may vary slightly depending on the particular method used to generate the CAPTCHA. When we compressed a text CAPTCHA using popular GIF format the resulting files were consistently 8 KB.

[15] Motoyama, et al. estimated that spammers paid humans \$1 to solve a thousand CAPTCHAs [31]