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

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 [43]. Offline dictionary attacks against passwords are — unfortunately — powerful and commonplace [43]. Adversaries have been able to compromise servers at large companies (e.g., Zappos, LinkedIn, Sony, Gawker [35, 36, 37, 38, 39]) resulting in the release of millions of cryptographic password hashes [39]. It has been repeatedly demonstrated that users tend to select easily guessable passwords [22, 28, 31, 40], and password crackers are able to quickly break many of these passwords [22]. 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 [36] — and as more and more training data — leaked passwords from prior breaches — becomes available [22]. 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 [40]. 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

---

*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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM 978-1-4503-2488-5/13/11 ...$15.00.

http://dx.doi.org/10.1145/2517312.2517319

---

In a few of these cases [35, 39] the passwords were stored in the clear.
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 4 we demonstrate how our GOTCHA could be integrated into an authentication protocol. We present the results from our user study in section 5, and in section 6 we challenge the AI and security communities to break our GOTCHA construction. In section 6 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 7.

1.1 Related Work

Inkblots 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. (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. — 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) generate garbled text, which is easy for a human to read, but difficult for a computer to decipher. Other versions of CAPTCHAs rely on the natural human capacity for audio or image recognition.

\[\text{Admittedly some people would dispute the use of the label 'easy.'}\]
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\cite{14}. However, if the adversary has access to the cryptographic hash of the password $h(pw)$ the server stores $(h(pw, r), r)$ for a random string $r$ 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\cite{49}.

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\cite{28}. The BCRYPT\cite{33} 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)\cite{28}. However, empirical studies show that user’s are often frustrated by restricting policies and commonly forget their passwords\cite{23, 24}. Furthermore, the cost of these restrictive policies can be quite high. For example, a Gartner case study\cite{44} estimated that it cost over $17 per password-reset call. Florencio and Herley\cite{47} 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\cite{1} and HosPs\cite{13}. 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 $H$ to denote the set of human users and $H(c, \sigma_t)$ to denote the response that a human $H \in H$ gives to the challenge $c \in \mathcal{C}$ at time $t$. Here, $\sigma_t$ denotes the state of the human’s brain at time $t$. $\sigma_t$ 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 $t$).

\footnote{In fact the resulting passwords are sometimes more vulnerable to an offline attack!\cite{29, 30}}
cause $\sigma_t$ 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 $A$ to denote the range of possible responses (answers) that a human might give to the challenges.

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

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

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

One of the requirements of a HOSP puzzle system is that the human $H$ must be able to consistently answer any challenge that is generated (e.g., $\forall c \in C$, $H$ can consistently solve $c$). These requirements seem to rule out promising ideas for HOSP constructions like Inkblots. 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 $t$.

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 between trials. By contrast, our GOTOCHA 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 $\vec{c}_\pi = (\vec{c}, \vec{a}) \in C$ of size $k$ is given by a $k$-tuple of challenges $\vec{c} = (c_1(\pi), \ldots, c_k(\pi)) \in C^k$ and solutions $\vec{a} = H((c_1(\pi), \ldots, c_k(\pi)), \sigma_0)$. The response to a matching challenge is a permutation $\pi' = H(\vec{c}_\pi, \sigma_0)$.

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

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

$d_k(\pi_1, \pi_2)$ 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 $\vec{c}_\pi$ with accuracy $\alpha$ if $\forall t \in \mathbb{N}$, $d_k(H(\vec{c}_\pi, \sigma_t), \pi) \leq \alpha$. We use $(\pi' \mid d_k(\pi, \pi') \leq \alpha)$ to denote the set of permutations $\pi'$ that are $\alpha$-close to $\pi$.

The puzzle generation process for a GOTOCHA 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 $\vec{c}$. 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 $r_1, r_2 \in \{0,1\}^k$ and outputs $k$ challenges $(c_1, \ldots, c_k) \leftarrow G_1(1^k, r_1, r_2)$.

$G_2$ is a randomized challenge generator that takes as input $1^k$ (security parameter), a random bit string $r_1 \in \{0,1\}^k$, and proposed answers $\vec{a} = (a_1, \ldots, a_k)$ to the challenges $G_1(1^k, r_1, r_2)$ and outputs a challenge $\vec{c} \leftarrow G_2(1^k, r_1, \vec{a})$. We say that the puzzle-system is $(\alpha, \beta)$-usable if

$$\Pr_{\vec{a} \sim \{0,1\}^k \mid \vec{c} \sim C} [\text{Accurate}(H, \vec{c}, \alpha)] \geq \beta,$$

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

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 $(I_1, \ldots, I_k)$ using random bits $r_1$, selects a random permutation $\pi : [k] \to [k]$ using random bits $r_2$, and returns $(I_{\pi(1)}, \ldots, I_{\pi(k)})$. The human’s response to an Inkblot $I_j$ is a randomized challenge generator $G$ — 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 4). Finally, $G_2$ might generate Inkblots $\vec{c} = (I_1, \ldots, I_k)$ using random bits $r_1$, and return the matching challenge $\vec{c}_\pi = (\vec{c}, \vec{a})$. In this case the matching challenge is for the user to match his labels with the appropriate Inkblot images to recover the permutation $\pi$. Observe that the final challenge $\vec{c}_\pi$ 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 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 $(\pi_{\text{a}(1)}, \ldots, \pi_{\text{a}(k)}) = H((I_{\pi_{\text{a}(1)}}, \ldots, I_{\pi_{\text{a}(k)}}, \sigma_0))$ with Inkblots $(I_{\text{a}(1)}, I_{\text{a}(2)})$ 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 $D_1$ and $D_2$ with advantage $\epsilon$ if

$$\left| \Pr_{\pi \sim \Pi_1} [A(x) = 1] - \Pr_{\pi \sim \Pi_2} [A(y) = 1] \right| \geq \epsilon.$$

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

4We 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.

5We note that a HOSP puzzle system $(G)$ can be modeled as a GOTOCHA puzzle system $(G_1, G_2)$ where $G_1$ does nothing and $G_2$ simply runs $G$ to generate the final challenge $\vec{c}$ 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 \((\alpha, \beta)\)-usable (2) Given a human \( H \in \mathcal{H} \) no probabilistic polynomial time algorithm can distinguish between distributions

\[
D_1 = \left\{ \left. \begin{array}{l}
H(G_1(1^x, r_1, r_2), \sigma_0)
\end{array} \right| r_1, r_2 \overset{\$}{\leftarrow} \{0, 1\}^n \right\}
\]

and

\[
D_2 = \left\{ \left. \begin{array}{l}
G_2(1^x, r_1, H(G_1(1^x, r_1, r_2), \sigma_0))
\end{array} \right| r_1 \overset{\$}{\leftarrow} \{0, 1\}^n \right\}
\]

with advantage greater than \( \epsilon \), 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

\[
D_3 = \left\{ \left. \begin{array}{l}
H(G_1(1^x, r_1, r_2), \sigma_0)
\end{array} \right| r_1, r_2 \overset{\$}{\leftarrow} \{0, 1\}^n \right\}
\]

and

\[
D_4 = \left\{ \left. \begin{array}{l}
G_2(1^x, r_1, H(G_1(1^x, r_1, r_2), \sigma_0))
\end{array} \right| r_1, r_2 \overset{\$}{\leftarrow} \{0, 1\}^n \right\}
\]

with advantage greater then \( \delta \).

### 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 \( h : \{0, 1\}^* \rightarrow \{0, 1\}^n \) and a random bit string \( s \in \{0, 1\}^* \) [17]. For example, if a user \((u)\) chose the password \( pw \) the server would store the tuple \((u, s, h(s, pw))\) (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

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

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 \( c_{\text{Bcrypt}} \) to denote the cost of querying the function \( \text{VerifyHash} \) and we use \( n_H \) (resp. \( n_{\text{Bcrypt}} \)) to denote the number of queries to the human (resp. \( \text{VerifyHash} \)). Queries to the human GOTCHA solver are much more expensive than queries to the cryptographic hash function \( h(c_H \gg c_{\text{Bcrypt}}) \) [33]. For technical reasons we limit our analysis to conservative adversaries.

**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 \( \text{VerifyHash} \) which is then used as a black box by \( Adv \) ), (2) The pseudorandom function \( \text{Extract} \) 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,

\[
\text{Cost}(Adv, D) = (n_H c_H + n_{\text{Bcrypt}})
\]

to denote the cost of the queries that the adversary makes to check each guess in \( D \), and \( \text{Succeed}(Adv, D, pw) \) to denote the event that the adversary makes a query to \( \text{VerifyHash} \) 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 \((C, \gamma, D)\)-successful if

\[
\Pr_{pw \in D}[\text{Succeed}(Adv, D, pw)] \geq \gamma.
\]

Our attack model is slightly different from the attack model in [17]. 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 [17], 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 \( \mathcal{I} \) to generate Inkblot images. Algorithm \( \mathcal{I} \) takes as input random bits \( r_1 \) and a security parameter \( k \) — which specifies the number of Inkblots to output. Algorithm \( \mathcal{I} \) makes use of the randomized subroutine \( \text{DrawRandomEllipsePairs} \) \([I, t, w, h, \text{height}]\) 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 of rotation, and the second ellipse is mirrored on the right half of the image. Figure \( \mathcal{I} \) is an example of an Inkblot image generated by algorithm \( \mathcal{I} \).

Our candidate GOTCHA is given by the pair \((G_1, G_2)\) — algorithms \( \mathcal{I} \) and \( \mathcal{I} \). \( G_1 \) runs algorithm \( \mathcal{I} \) 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  
\textbf{▷} The following operations only use the random bit string $r_1$ as a source of randomness  
\textbf{DrawRandomEllipsePairs}($I_j$, 150, 60, 60) 
\textbf{DrawRandomEllipsePairs}($I_j$, 70, 20, 20) 
\textbf{DrawRandomEllipsePairs}($I_j$, 150, 60, 20) 
\textbf{return} $(I_1, \ldots, I_k)$  
\textbf{▷} Inkblot Images

Inkblot images, and then returns these images in permuted order — using a function 
\textbf{GenerateRandomPermutation}($k$, $r$), which generates a random permutation $\pi : [k] \rightarrow [k]$ using random bits $r$. $G_2$ also runs algorithm 3 to generate $k$ Inkblot images, and then outputs a matching challenge.

**Algorithm 2 G1**

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

After the Inkblots $(I_{\pi(1)}, \ldots, I_{\pi(k)})$ have been generated, the human user is queried to provide labels $\ell_{\pi(1)}, \ldots, \ell_{\pi(k)}$ where 
$\langle \ell_{\pi(1)}, \ldots, \ell_{\pi(k)} \rangle = H(\langle I_{\pi(1)}, \ldots, I_{\pi(k)} \rangle, \sigma_0)$.

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 G1 — is to match the Inkblot images $I_1, \ldots, I_k$ with the user generated labels $\ell_1, \ldots, \ell_k$ to recover the permutation $\pi$.

**Algorithm 3 GenerateMatchingChallenge G2**

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

**Observation:** Notice that if the random bits provided as input to \textbf{GenerateInkblotImages} and \textbf{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: \textbf{Create Account} (protocol G2) and \textbf{Authenticate} (protocol G2). 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 $(\ell_1, \ldots, \ell_k) = H(\langle I_1, \ldots, I_k \rangle, \sigma_0)$ based on his mental state at the time — the server stores these labels in permuted order $\langle \ell_{\pi(1)}, \ldots, \ell_{\pi(k)} \rangle$. To authenticate later the user will have to match these labels with the corresponding Inkblot images to recover the permutation $\pi$.

In section 3 we argue that the adversary who wishes to mount a cost effective offline attack needs to obtain constant feedback from a human. Following [12] 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 $(I_1, \ldots, I_k)$ to the user where:  
$r' \leftarrow \{0,1\}^n, r_1 \leftarrow$ Extract $(pw, r')$, $r_2 \leftarrow \{0,1\}^n$ and $(I_1, \ldots, I_k) \leftarrow$ \textbf{GenerateInkblotImages}$(1^k, r_1)$ 
**(User):** Sends responses $(\ell_1, \ldots, \ell_k)$ back to the server where:  
$(\ell_1, \ldots, \ell_k) \leftarrow H(\langle I_1, \ldots, I_k \rangle, \sigma_0)$.  
**(Server):** Store the tuple $t$ where $t$ is computed as follows:  
Salt: $s \leftarrow \{0,1\}^n$  
$\pi \leftarrow$ \textbf{GenerateRandomPermutation}($k$, $r_2$).  
$h_{pw} \leftarrow h(u, s, pw, \pi(1), \ldots, \pi(k))$ 
$t \leftarrow (u, r', s, h_{pw}, \ell_{\pi(1)}, \ldots, \ell_{\pi(k)})$

**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 $c$ to the user where $c$ is computed as follows:  
Find $t = (u, r', s, h_{pw'}, \ell_{\pi(1)}, \ldots, \ell_{\pi(k)})$  
r\textsuperscript{*} \leftarrow $\text{Extract}(pw', r')$ 
$I_1, \ldots, I_k \leftarrow$ \textbf{GenerateInkblotImages}$(1^k, r_1)$  
$c_\hat{a} \leftarrow (\langle I_1, \ldots, I_k \rangle, (\ell_{\pi(1)}, \ldots, \ell_{\pi(k)}))$ 
**(User):** Solves $c_\hat{a}$ and sends the answer $c' = H(\hat{c}, \sigma_1)$.  
**(Server):**  
\textbf{for all} $\pi_0 \text{ s.t. } d_k(\pi_0, \pi') \leq \alpha$  
\textbf{do}  
$h_{pw,0} \leftarrow h(u, s, pw', \pi_0(1), \ldots, \pi_0(k))$  
\textbf{if} $h_{pw,0} = h_{pw}$ \textbf{then} Authenticate

Deny

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

---

7For a general GOTCHA, protocol [13] 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 $H(G_2(\cdot), \sigma_0)$. Protocol [13] takes advantage of the fact that $\pi = H(G_2(\cdot), \sigma_0)$ is already known.
in protocol $\text{α}$. 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 $\text{β}$. 

Claim $\text{γ}$ says that a legitimate user can successfully authenticate if our Inkblot construction satisfies the usability requirements of a GOTCHA. The proof of claim $\text{γ}$ can be found in appendix $\text{δ}$.

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

One way to improve usability of our authentication protocol is to increase the neighborhood of acceptably close matchings by increasing $\alpha$. The disadvantage is that the running time for the server in protocol $\gamma$ increases with the size of $\alpha$. Claim $\text{ε}$ bounds the time needed to enumerate over all close permutations. The proof of claim $\text{ε}$ can be found in appendix $\text{ζ}$.

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

$$\left|\left\{ \pi' \big| d_k (\pi', \pi) \leq \alpha \right\}\right| \leq 1 + \alpha \left( \begin{array}{c} k \\ i \end{array} \right)! .$$

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$^6$. Organizations are already advised to use password hash functions like BCRYPT $\text{η}$ 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 $\text{ξ}$ 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$^8$. 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

---

<table>
<thead>
<tr>
<th>α-accurate # participants</th>
<th>Phase 1</th>
<th>Phase 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>α = 0</td>
<td>17</td>
<td>0.29</td>
</tr>
<tr>
<td>α = 2</td>
<td>22</td>
<td>0.38</td>
</tr>
<tr>
<td>α = 3</td>
<td>26</td>
<td>0.45</td>
</tr>
<tr>
<td>α = 4</td>
<td>34</td>
<td>0.59</td>
</tr>
<tr>
<td>α = 5</td>
<td>40</td>
<td>0.69</td>
</tr>
</tbody>
</table>

---

**Table 1:** Completion Times

---

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

**Table 2:** Usability Results: Fraction of Participants who would have authenticated with accuracy parameter $\alpha$ 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 $\text{η}$. 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 $\text{ζ}$.

### Discussion.

Our user study provides evidence that our construction is at least (0, 0.2)-usable or (5, 0.5)-usable. While this means that our Inkblot Matching GOTCHA could be used by a significant fraction of the population to protect their pass-
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. 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 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 while wearing a voodoo mask” were often mismatched, while more elaborate stories increased the likelihood of correct matching). To generate Inkblots that can be labeled more quickly and matched more accurately?

### 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 $(pw_1, ..., pw_10) \in \{0,10^8\}$ and $pw_2 \in \{0,10^8\}$. We used a function $\text{GenerateInkblots}(pw_1, 10)$ to generate ten Inkblots $I_1, ..., I_{10}$ for each password, and we had a human label each Inkblot image $(I_1, ..., I_{10}) \leftarrow H((I_1, ..., I_{10}), s_0)$. We selected a random permutation $\pi : [10] \rightarrow [10]$ for each account, and generated the tuple

$$T_i = \left( s_i, h(pw_1, s_i, \pi(1), ..., \pi(10)), \ell_{\pi(1)}, ..., \ell_{\pi(10)} \right),$$

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 $T_1, ..., T_10$ — 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 $pw'$ and evaluate $h(pw', s_i, \pi(1), ..., \pi(10))$ for every possible permutation $\pi : [10] \rightarrow [10]$. 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) — 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 and ) significantly mitigates the threat of offline attacks. An informal interpretation of our main technical result — Theorem — 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 $\epsilon$ and $\delta$ are not negligibly small.

**Theorem 1.** Suppose that our user selects his password uniformly at random from a set $D$ (e.g., $pw \leftarrow D$) and creates his account using protocol . If algorithms and are an $(\epsilon,\delta,\mu)$-GOTCHA then no conservative offline adversary is $(\gamma,\epsilon+\delta)$-GOTCHA.

Proof of Theorem 1. (Sketch) We use a hybrid argument. An adversary who breaches the server is able to recover the tuple $t = (u, s, h(u, s, pw, \pi(k)), \ell_{\pi(k)}(1), ..., \ell_{\pi(k)}(n))$ 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 $t_0 = (u, r, s, h(u, s, pw, \pi(1), ..., \pi(k)), \ell_{\pi(1)}(1), ..., \ell_{\pi(k)}(n))$ 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

\[ h^{2^{15}} \]
VerifyHash \( (pw', \pi') = \begin{cases} 
1 & \text{if } pw' = pw \text{ and } \pi' = \pi \\
0 & \text{otherwise} 
\end{cases} 
\)

and uses VerifyHash as a blackbox. We say that Adv queries a human \( H \) about password \( pw' \) if it queries \( H \) for \( H(\text{GenerateInkblotImages}(1^k, \text{Extract}(pw', r'))) \) 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

\[
\text{VerifyHash}^1 (pw', \pi') = \begin{cases} 
1 & \text{if } pw' \notin D', pw' = pw \text{ and } \pi' = \pi \\
0 & \text{otherwise} 
\end{cases} 
\]

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

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

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

4. World 3: \( W_3 \) denotes a hypothetical real world which is similar to world 2, except that the labels \( \ell_1^{(1)}, \ldots, \ell_k^{(1)} \) are replaced with the labels \( \ell_1^{(2)}, \ldots, \ell_k^{(2)} \), where \( \pi': [k] \to [k] \) is a new random permutation, and the labels \( \ell_i' \) are for a completely unrelated set of Inkblot challenges

\[
\ell_1, \ldots, \ell_k \leftarrow H (G_1 \left( 1^k, x_1, x_2 \right)),
\]

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 \((C, \gamma, D)\)-successful for \( C < \gamma |D|^{2\mu(k) \cdot c_h} \), because the fake Inkblot labels are not correlated with the actual Inkblots that were generated with the real password. Our particular adversary cannot be \((C, \gamma, D)\)-successful for \( C < \gamma |D|^{2\mu(k) \cdot c_h} + |D'|^{c_h} \). In world 2 the adversary might improve his chances of success by looking at the Inkblot labels, but by definition of \((\alpha, \beta, \epsilon, \delta, \mu)\)-GOTCHA his chances change by at most \( \delta \). 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 \( \epsilon \). 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 human feedback during puzzle generation — unlike HOSPs\[^1\] — 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\[^2\]).

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 obfuscator\[^3\]. However, the approach may not be entirely hopeless. Point functions\[^4\] 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\[^5\] and to obfuscate \( NC^1 \) circuits\[^6\]. 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\[^10\], zero-knowledge proofs\[^11\], secure multiparty computation\[^12\]) and in complexity theory\[^13\], 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.\[^20\] — 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\[^21\].

6. REFERENCES


\[^1\] 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.

\[^2\] A polynomial time verifier can verify \( \text{PSPACE} \)-complete languages by interacting with a powerful prover\[^11\], by contrast the same verifier can only check proofs of \( \text{NP} \)-Complete languages without interaction.
APPENDIX

A. MISSING PROOFS

Reminder of Claim 1. If \( (G_1, G_2) \) is a \((\alpha, \beta, \epsilon, \delta, \mu)\)-GOTCHA then at least \( \beta \)-fraction of humans can successfully authenticate using protocol \( \mathcal{P} \) after creating an account using protocol \( \mathcal{Q} \).

Proof of Claim 1. A legitimate user \( H \in \mathcal{H} \) will use the same passwords in protocols \( \mathcal{P} \) and \( \mathcal{Q} \). Hence,

\[
r_1' = \text{Extract} (pw', r') = \text{Extract} (pw, r') = r_1,
\]

and the final matching challenge \( c_r \) is the same one that would be generated by \( G_2 (1^k, r_1, H (G_1 (1^k, r_1, r_2), \pi_0)) \). If \( c_r \) is consistently solvable with accuracy \( \alpha \) by \( H \) — by definition \( \mathcal{P} \) is the case for at least \( \beta \)-fraction of users — then it follows that

\[
d_k (\pi, \pi', \sigma_t) \leq \alpha,
\]

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

\[
h_{pw,0} = h (u, s, pw', \pi_0(1), ..., \pi_0(k)) = h (u, s, pw, \pi(1), ..., \pi(k)) = h_{pw},
\]

and protocol \( \mathcal{Q} \) accepts. \( \square \)

Reminder of Claim 2. For all permutations \( \pi : [k] \rightarrow [k] \) and \( \alpha \geq 0 \)

\[
|\{ \pi' | d_k (\pi, \pi', \sigma_t) \leq \alpha \}| \leq 1 + \alpha \sum_{i=0}^{\alpha} \binom{k}{i} !.
\]

Proof of 2. It suffices to show that \( \binom{k}{i} j ! \geq |\{ \pi' | d_k (\pi, \pi', \sigma_t) = j \}|. \)

We first choose the \( j \) unique indices \( i_1, \ldots, i_j \) on which \( \pi \) and \( \pi' \) 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. \( \pi (i_k) \neq \pi' (i_k) \) for all \( k \leq j \).

\( \square \)

B. HOSP: PRE-GENERATED CAPTCHAS

The HOSP construction proposed by \( \mathcal{P} \) 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 \( \mathcal{Q} \) 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 \( \mathcal{P} \). Let \( S \) be the space required to store one

\footnote{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 $D = \{P_1, \ldots, P_{2^n}\}$ 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 $\Rightarrow$ and $C_H = $ $0.001 $$. In this case $|D| = \frac{4 \times 10^9}{8 \text{ KB}} \approx 10^9$ 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 [3, 4, 5]. 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 [22].