23rd Annual Computer Security Applications Conference

Breaking Visual CAPTCHAs with Naïve Pattern Recognition Algorithms

Jeff Yan, Ahmad Salah El Ahmad
School of Computing Science, Newcastle University, UK
{Jeff.Yan, Ahmad.Salah-El-Ahmad}@ncl.ac.uk

Abstract

Visual CAPTCHAs have been widely used across the Internet to defend against undesirable or malicious bot programs. In this paper, we document how we have broken most such visual schemes provided at Captchaservice.org, a publicly available web service for CAPTCHA generation. These schemes were effectively resistant to attacks conducted using a high-quality Optical Character Recognition program, but were broken with a near 100% success rate by our novel attacks. In contrast to early work that relied on sophisticated computer vision or machine learning algorithms, we used simple pattern recognition algorithms but exploited fatal design errors that we discovered in each scheme. Surprisingly, our simple attacks can also break many other schemes deployed on the Internet at the time of writing: their design had similar errors. We also discuss defence against our attacks and new insights on the design of visual CAPTCHA schemes.

1 Introduction

CAPTCHA stands for "Completely Automated Public Turing Test to Tell Computers and Humans Apart", and a CAPTCHA is a program that generates and grades tests that most human can pass, but current computer programs cannot pass [1]. Such tests are often called CAPTCHA challenges, and they are based on a hard, open problem in AI. At present, CAPTCHA is almost a standard security mechanism for defending against undesirable and malicious bot programs on the Internet, e.g., bots that could sign up for thousands of accounts a minute with free email service providers, bots that could send out thousands of spam messages each minute, and bots that could in one act post numerous comments in weblogs ("blogs") pointing both readers and search engines to irrelevant sites.

To date, the most commonly used are various types of *visual* CAPTCHAs, in which a challenge often appears as an image of distorted text that the user must decipher. These schemes typically make use of the difficulty of recognising distorted text with state of the art computer programs. Well-known visual CAPTCHAs include EZ-Gimpy, Gimpy and Gimpy-r (see [1, 22]), all developed at Carnegie Mellon University. Google and Microsoft also developed their own visual CAPTCHAs for their web email services. Many more schemes have been put into practice, although they are less visible in the literature. In recent years, there were some pioneering research efforts exploring how to design visual CAPTCHAs properly, e.g. [2, 4, 5, 7, 24]. However, it appears that this subject is still an art, rather than a science.

Captchaservice.org is, to our knowledge, the first web service that is designed for the sole purpose of generating CAPTCHA challenges. It supports multiple visual and non-visual CAPTCHA schemes, and it is publicly available at [16]. Using the API provided by this service, people can obtain a chosen type of CAPTCHA challenge generated on the fly to protect their blogs from comment spam attacks, or to defend against other type of bots. The design of this web service and the CAPTCHA schemes it supports were discussed and analysed in a recent peer-reviewed paper [8]. In the present paper, we document how, using novel attacks, we have broken most visual CAPTCHA schemes provided by *captchaservice.org*. These CAPTCHAs were effectively resistant to Optical Character Recognition (OCR) software attacks, and thus appeared to be secure. However, our attacks could achieve a success rate of near 100% for each of these schemes, and each challenge could be broken essentially instantly. As a result, an adversary could easily bypass all the defence provided by these schemes in real time.

Breaking CAPTCHAs[1] is not new. For example, Mori and Malik [10] have broken the EZ-Gimpy (92% success) and the Gimpy (33% success) CAPTCHAs with sophisticated object recognition algorithms. Moy et al [11] developed distortion estimation techniques to break EZ-Gimpy with a success rate of 99% and 4-letter Gimpy-r with a success rate of 78%. In contrast to such earlier work that relied on sophisticated computer vision algorithms, our attacks used naïve pattern recognition algorithms but ex-

[1] The term of "Breaking CAPTCHA" is ambiguous. For example, when CAPTCHA is interpreted as a simple challenge-response protocol, "breaking CAPTCHA" can mean breaking the protocol, e.g. via a man-in-the middle or an oracle attack. In this paper, "breaking CAPTCHA" means to write a computer program that automatically solves CAPTCHA challenges – ideally, this task should be as hard as solving the underlying AI problem. ("Breaking CAPCHA", "breaking a CAPTCHA protocol", and "Defeating CAPTCHA based bot defence" are three different but related notions, as clarified in [15].)

ploited fatal design errors in each of the schemes that we have broken. Compared with the vision researchers who aimed to advance research in AI by breaking CAPTCHAs, we as computer security specialists aimed to understand how a CAPTCHA could fail as a security system and what we could learn from these failures.

The main contributions of this paper are the following. First, we have identified fatal flaws in the design of four visual CAPTCHA schemes from *captchaservice.org*, and shown that all these schemes can be broken with a high success rate. We have also found that these flaws are also present in many other CAPTCHA schemes deployed on the Internet, also making them vulnerable to our attacks. We informed CERT of our results several months ago so that the developers of the schemes were given ample opportunity to improve their CAPTCHA security by fixing these flaws. Second, our work reveals that the security of CAPTCHAs is much poorer in real life than it might have appeared to be. Many schemes deployed for everyday use on the Internet at the time of writing were very weak, and they could be easily broken without the need to invent a sophisticated algorithm. Moreover, we also discuss lessons we have learnt from breaking these schemes and how to defend against our attacks – all this contributes to understanding how to design better visual CAPTCHA schemes in general. Our work reiterates the necessity of independent security evaluations before a system is considered secure. Without such evaluation, a system might result in providing a false sense of security. This also raises the following important questions: How do we turn the design of CAPTCHAs from an art to a science, and in particular, how do we rigorously evaluate the robustness (and other properties) of a CAPTCHA scheme?

The rest of this paper is organised as follows. Section 2 discusses related work. Section 3 reviews the schemes we have broken, and evaluates their strength with a high quality OCR program. Sections 4-6 present our attacks on each scheme respectively. Section 7 measures the speed of our attacks. Section 8 mentions other schemes that are vulnerable to our attacks, and discusses both lessons we have learnt, and defences to our attacks. Section 9 gives concluding remarks.

2 Related Work

Chellapilla and Simard [7] attempted to break a number of visual CAPTCHAs taken from the web (including those used in Yahoo and Google/Gmail) with machine learning algorithms. However, their success rates were low, ranging from merely 4.89% to 66.2%. No empirical data for attack speed were reported, and therefore it is unclear whether their attacks could break these schemes in real time. An attack on an unnamed simple CAPTCHA scheme with neural networks was discussed at [6], and it achieved a success rate of around 66%.

PWNtcha [9] is an excellent web page that aims to "demonstrate the inefficiency of many captcha implementations". It comments briefly on the weaknesses of a dozen visual CAPTCHAs. These schemes were claimed to be broken with a success rate ranging from 49% to 100%. However, no technical detail was publicly available (and probably as a consequence, at a prominent place of this web page, a disclaimer was included that it was not "a hoax, a fraud or a troll"). More distantly related (in spirit) is work by Naccache and Whelan [12] on decrypting words that were blotted out in declassified US intelligence documents, although it was not about CAPTCHAs as such.

The limitations of defending against bots with CAPTCHAs (including protocol-level attacks) were discussed in [15]. A recent survey on CAPTCHAs research can be found in [14].

3 Targeted CAPTCHA schemes

Captchaservice.org supports the following four visual schemes:

- *word_image*: In this scheme, a challenge is a distorted image of a six-letter word.
- *random_letters_image*: A challenge is implemented as a distorted image of a random six-letter sequence.
- *user_string_image*: A challenge is a distorted image of a user-supplied string of at most 15 characters.
- *number_puzzle_text_image*: This is a multi-modal scheme, which includes a distorted image of a random number, as well as a textual description of a puzzle involving the number. A user can solve such a challenge either by recognising the number in the image, or by solving the textual puzzle. The advantage of such a multimodal scheme is mainly to improve its usability and accessibility. In this paper, we are interested in its visual mode only.

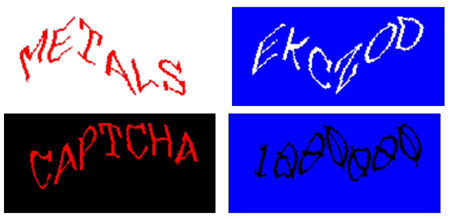


Fig 1. Sample challenges for four visual CAPTCHAs available at Captchaservice.org (clockwise: ***word_image*****,** ***random_letters_image*****,** ***user_string_image*****,** ***number_puzzle_text_image*****)**

All these schemes use a *random_shear distortion* technique, which [8] describes, thus: "the initial image of text is distorted by randomly shearing it both vertically and horizontally. That is, the pixels in each column of the image are translated up or down by an amount that varies randomly yet smoothly from one column to the next. Then the same kind of translation is applied to each row of pixels (with a smaller amount of translation on average)." Fig 1 shows sample challenges from the above schemes. The *word_image* scheme also supports an additional distortion technique, but it is beyond the scope of this paper.

To benchmark how resistant they were to OCR software attacks, we tested all except the third scheme with ABBYY FineReader V.8 [17], a commercial OCR product. We chose this program for two reasons: 1) as an award-winning product, it is considered one of the best in the market, and 2) we happen to have access to the software. We did not test the *user_string_image* scheme, since other than that a user could specify the text string, it seemed that nothing else was different from the first or second scheme.

CAPTCHA Scheme	**Number of challenges**						
	All characters Recognised	**Partially recognised (no. of characters)**					**Zero characters Recognised**
		5	**4**	**3**	**2**	**1**	
Word_image	0	0	3	6	8	16	67
Random_letters_image	0	0	2	4	8	20	66
Number_puzzle_text_image	10	13					77

Table 1. Test results of resistance to OCR software automatic recognition attacks.

We collected 100 random samples from [16] for each scheme to be tested, and performed the following two attacks on them: 1) we fed each sample into the OCR software for an automated recognition, and 2) we manually segmented each sample, and then let the software recognise individual characters. The test results are as follows (Table 1 summarises the results of Attack 1).

Word_image. In Attack 1, none of the samples was completely recognised. For 67 challenges, none of the characters were recognised, whereas the remaining 33 challenges were partially recognised between 1 and 4 characters. Fig 2 (a) gives snapshots of a partially recognised challenge ("REMOTE" recognised as "R£MO^") and a completely failed one ("FRISKY" recognised as "tmsi"). In Attack 2, 38% (128 out of 600) letters were recognised; however, only one sample had all its 6 letters recognised and another had five of its letters recognised. This is not surprising, since the individual recognition rate theoretically implies a mere success rate of 0.3% (.38^6) for breaking this scheme.

Random_letters_image. In Attack 1, no sample was completely recognised. 66 challenges had none of their characters recognised, but the remaining 34 challenges were partially recognised between 1 and 4 characters. Fig 2(b) gives snapshots of a partially recognised challenge and a completely unrecognised one. In Attack 2, 41% (248 out of 600) letters were recognised. In theory, this implies a success rate of about 0.5% (.41^6) for breaking this scheme. In our experiment, 2% (2 out of 100) samples had all its 6 letters recognised.

Number_puzzle_text_image. In Attack 1, 10 samples were completely recognised, 13 partially recovered and 77 challenges having none of their characters successfully recognised. A close look at this set of samples showed: they had a varying length, consisting of 1-7 digits (on average 2.9 digits per challenge). So we did not pinpoint how many characters were recovered for those partially recognised samples. We also found that all those completely recovered challenges contained a single digit only, like the first sample in Fig 2(c). While the OCR program recovered such simple challenges, it completely failed to recognise any of more complex ones, such as the other sample in Fig 2(c). In Attack 2, 16% (46 out of 286) characters were successfully recognised. However, only 11 samples were completely recognised, 30 partially recovered and 59 challenges having none of their characters successfully recognised.

Therefore, **it appears that the *random_shear* distortion provides reasonable resistance to OCR software attacks, and the distortion implemented in the *word_image* scheme is as good as in the *random_letters_image* scheme**. We also observed that the OCR program we used in general had difficulty in differentiating between characters 'H' and 'N', 'I' and 'T', 'L' and 'V', and 'M' and 'W' in the samples we presented.

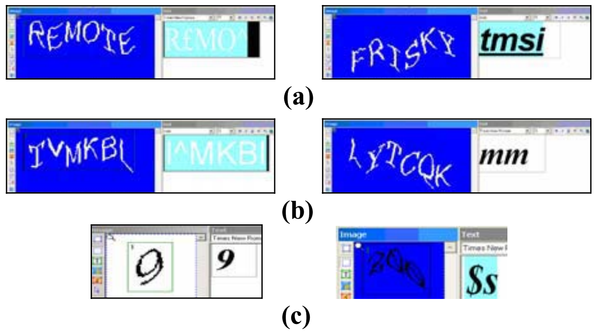


Fig 2. Snapshots of the results of OCR Attack 1. (a) Two *word_image* samples. (b) Two *random_letters_image* samples. (c) Two *number_puzzle_text_image* samples.

4 Breaking Scheme 1

Although the visual CAPTCHA schemes discussed in the previous section appeared to be secure, we took up the challenge of breaking them. In this and following sections, we report the details of defeating the *word_image*, *random_letters_image*, *number_puzzle_text_image* and *user_string_image* schemes, which are labelled as Scheme 1, 2, 3 and 4, respectively.

In Scheme 1, each challenge was implemented as a distorted image of a six-letter English word, which was randomly chosen from a fixed set of 6,000 words [8]. More sample challenges (taken from [16]) are shown in Fig 3. We have broken this scheme with a basic attack algorithm and a number of refinements.

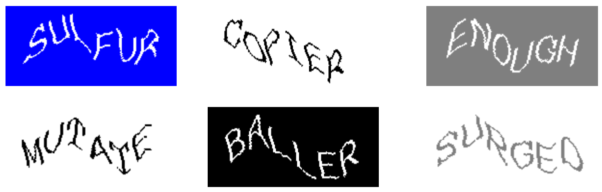


Fig 3. Scheme 1 CAPTCHA sample challenges (Each image is of 178 × 83 pixels, PNG format).

4.1 Basic attack algorithms

With the aid of [8] and documentation available at [16], we studied a sample set of 100 random *word_image* CAPTCHA challenges which we collected, and established the following empirical observations.

- Only two colours were used in each challenge, one for background and another for foreground which was the distorted challenge text; the choice of colours was either random or specified by the user. Therefore, it is easy to separate the text from the background.

Letter	Pixel Count
A	183
B	217
C	159
D	192
E	163
F	133
G	190
H	186
I	121
J	111
K	178
L	111
M	233
N	239
O	178
P	162
Q	229
R	208
S	194
T	175
U	164
V	162
W	234
X	181
Y	153
Z	193

Table 2. A letter–pixel count lookup table for letters A-Z. (Note: 'J' and 'L' have the same pixel count. So are 'K' and 'O', and 'P' and 'V'.)

- Only capital letters were used. Although a letter might be distorted into a different shape each time, it consisted of a constant number of foreground pixels in a challenge image. That is, a letter had a constant *pixel count*. We worked out the pixel count for each of the letters A to Z (see Table 2). As plotted in Fig 4, most letters had a distinct pixel count.

- Few letters overlapped or touched with each other in a challenge, so many challenges were vulnerable to a vertical segmentation attack: the image could be vertically divided by a program into segments each containing a single character.

Our basic attack algorithm is largely based on the above observations. One of its key components is a vertical segmentation algorithm, which works as follows.

1. *Obtaining the top-left pixel's colour value,* which defines the background colour of an image. Any pixel of a different colour value in this image is in foreground, i.e. part of the distorted text.
2. *Identifying the first segmentation line.* We map the image into a coordinate system, in which the top-left pixel has coordinates (0, 0), the top-right pixel (image width, 0) and the bottom-left pixel (0, image height). Starting from point (0, 0), a vertical "slicing" process traverse pixels from top to bottom and then from left to right. This process stops once a pixel with a non-background colour is detected. The X co-ordinate of this pixel, x_1, defines the first vertical segmentation line $X = x_1 - 1$.

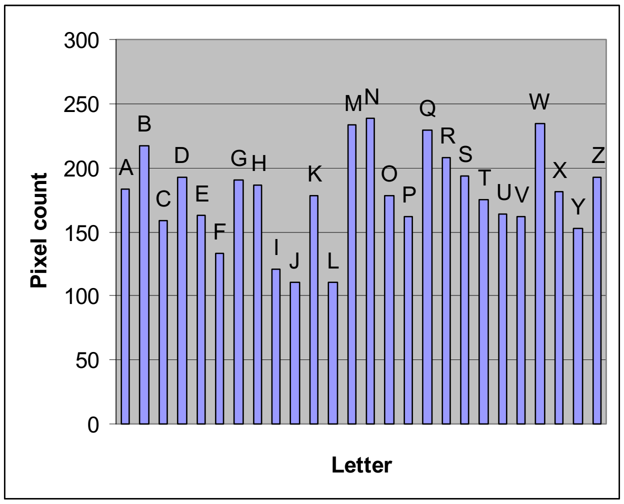


Fig 4. Letters A-Z and their pixel counts.

3. Vertical slicing continues from $(x_1+1, 0)$, until it detects another vertical line that does not contain any foreground pixels – this is the next segmentation line.
4. Vertical slicing continues from a pixel to the right of the previous segmentation line. However, the next vertical line that does not contain any foreground pixel is not necessarily the next segmentation line. It could be a redundant segmentation line, which would be ignored by our algorithm. Therefore, only when the vertical slicing process cuts through the next letter, the next vertical line that does not contain any foreground pixels is the next segmentation line.

5. Step 4 repeats until the algorithm determines the last segmentation line (after which, the vertical slicing will not find any foreground pixels).

Once a challenge image is vertically segmented, our attack program simply counts the number of foreground pixels in each segment. Then, the pixel count obtained is used to look up Table 2, telling the letter in each segment.

Fig 5 shows how our basic attack has broken a challenge. First, the vertical segmentation divided the challenge into 6 segments. Second, each segment was scanned to get the number of foreground pixels in it. Then, the pixel count obtained in the previous step was used to look up the mapping table, recognising a character C_i for each segment S_i (i=1, …, 6). Finally, the string 'C_1 C_2 C_3 C_4 C_5 C_6' gives the result.

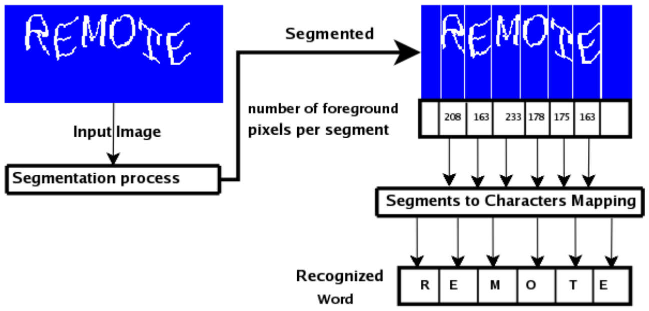


Fig 5. The basic attack: an example.

By combining the vertical segmentation and a lookup table, we achieved on the sample set a success rate of 36%, that is, 36 out of 100 challenges were completely broken.

4.2 Enhancement 1: dictionary attack

Our basic algorithm would fail to break some challenges completely. Fig 6 gives a failing example, where the vertical segmentation method could not separate letters 'S' and 'K' because the vertical slicing line could not split the two letters without touching both of them. Our basic attack could not do anything more than to give a partially recognised result "FRI**Y" (we use '*' to represent one unrecognised character). However, since Scheme 1 challenges all used words, our basic attack was enhanced by the following "dictionary attack".

A dictionary of about 6,000 six-letter English words was introduced. Since the dictionary used in Scheme 1 was not available, as a starting point we compiled our dictionary using a free wordlist collection [13] that is often used with password crackers.

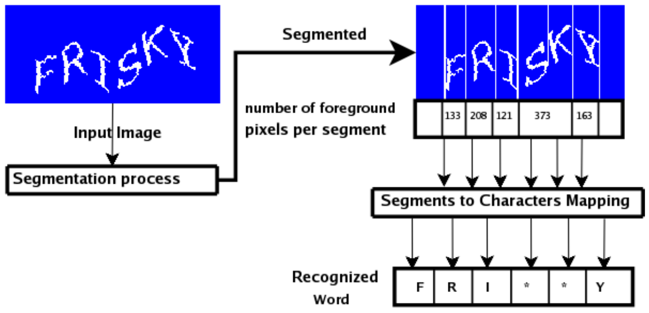
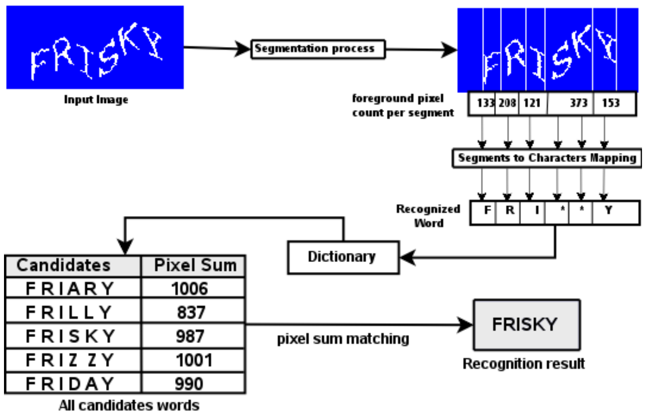


Fig 6. The basic attack: a failing example. (The partial line between 'S' and 'K' is for illustration only, and it did not exist in a segmented result.)

Any partial result returned by the basic algorithm was used as a string pattern to identify candidate words in the dictionary that match the pattern. Since there could be multiple candidate words, a simple solution was introduced to find the best possible result as follows. For each dictionary entry, we pre-computed (using Table 2) a *pixel sum*, which is the total number of pixels this word could have when it was embedded in a CAPTCHA challenge. This pixel sum was stored along with the word in the dictionary. We also worked out, on the fly, a pixel sum for the unbroken challenge, which is the total number of all foreground pixels in the challenge. The first candidate word with the same pixel sum as the challenge was returned as the final recognition result.



Candidates	Pixel Sum
FRIARY	1006
FRILLY	837
FRISKY	987
FRIZZY	1001
FRIDAY	990

Fig 7. The basic attack: two enhancements.

Fig 7 illustrates how the enhanced algorithm worked. In this case, the partial result 'FRI**Y' obtained by the basic algorithm was used to identify all words that start with 'FRI' and end with 'Y' in the dictionary. Five candidate words were found: 'FRIARY', 'FRILLY', 'FRISKY', 'FRIZZY' and 'FRIDAY'. However, 'FRISKY' was returned as the best possible result, since it was the only candidate having a pre-computed pixel sum of 987, which equals to the pixel sum of the unbroken challenge (133+208+121+372+153=987).

To make the dictionary attack work properly, it is crucial to create a correct string pattern after the vertical segmentation process. For example, when the vertical segmentation divided an image into only four segments and the corresponding partial result was in the following form: 'B□B□', it was important to determine how many unrecognised letters were in each box '□'. Otherwise, 'B*B***', 'B**B**' or 'B***B*' would give totally different recognition results. If all these patterns were used to look up the dictionary, it would be likely to find many candidates with an identical pixel sum.

This is a problem of indexing letters in their correct positions, and it was addressed using the following two-step method.

1) For some cases, it was trivial to work out a string pattern with contextual information. For instance, if a segmented image contained only one unrecognised segment, e.g. the example in Fig 7, the number of unrecognised characters in the segment was six minus the number of all recognised characters. Another straightforward case was when no character was recognised in an image – then the number of unrecognised segments in the image did not really matter. For example, an image segmented into three unrecognised segments '□□□' would be no different to one for which the vertical segmentation completely failed.

2) When the above method did not work, e.g. in the case of 'B□B□', we relied on the number of pixels in each unrecognised segment in order to deduce how many characters the segment contained. For example, when the number of pixels in a segment was larger than 239 (the largest pixel count in Table 2, i.e., 'N') but smaller than 2×239, it was likely that this segment had two unrecognised letters. There were exceptions that could not be handled this way. Although the average pixel count for letters A-Z was 178.80, 'J', 'L' and 'I' had a pixel count much smaller than the average. For example, the pixel sum of 'ILL' or 'LIL' was only 343; the pixel sum of 'LI' or 'IL' was a mere 232. We used an exception list to handle such cases. On the other hand, the combination of 'LLL', 'JK' and 'KJ' never or rarely occurs in English words. Due to the paper space limit, more details of this method are skipped.

An alternative way of doing the pixel sum matching was to use unrecognised segment(s) only. In this way, no pixel sum would be stored in the dictionary, but more computation would have to be done on the fly.

It is also worth noting that when none of the letters in a challenge could be recognised by the basic algorithm, the pixel sum matching method in the dictionary attack could serve as the last resort.

4.3 Further enhancements

The following enhancements were developed to handle typical "troublemakers" that could not be broken by the above techniques.

Letters with an identical pixel count. Letters having an identical pixel count could confuse our basic algorithm. For example, the challenge in Fig 8 (a) was successfully segmented into 6 parts, but it was initially recognised as "OELLEY", leading to an incorrect result. Since 'O' and 'K' have the same pixel count, our basic algorithm had only a 50% of chance for breaking this challenge.

To overcome this problem, we relied on the following "spelling check": if a challenge includes a letter with a pixel count of 111 ('J' or 'L'), 178 ('K' or 'O'), or 162 ('P' or 'V'), each we generate variant, and then carry out multiple dictionary lookups to rule out candidate strings that are not proper words. For example, in the above case, both 'OELLEY' and 'KELLEY' were looked up in the dictionary. Since only "KELLEY" was in the dictionary, it was returned as the best possible result.

This "spelling check" technique was also used to enhance the string pattern matching in the dictionary attack. For example, if a partial result recognised by the basic algorithm was "V*B*IC", then both "V*B*IC" and "P*B*IC" would be valid matching patterns for identifying candidate words in the dictionary.

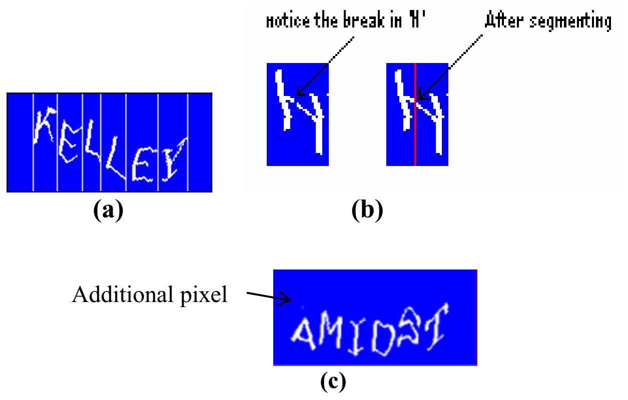


Fig 8. Typical troublemakers: a) Letters with an identical pixel count; b) Broken letters; c) Letters with additional or less pixels.

Broken characters. A few challenges contained broken letters that misled the segmentation algorithm. As shown in Fig 8 b), due to a break in 'H', the letter was segmented into two parts instead of one.

To overcome this problem, we introduced a two-step method as follows. First, once the vertical segmentation was done, our algorithm tested whether a segment was complete: if the number of foreground pixels in a segment was smaller than 111, the smallest pixel count in Table 2, then this segment was incomplete; if the number of foreground pixels in a segment was larger than 111 but smaller than 239 (the largest pixel count in Table 2, i.e., 'N') and this number could not be found in the lookup table, then this segment is incomplete. Second, an incomplete segment would be merged with its neighbouring segment(s). A proper merging of segment was one for which the combined pixel count could lead to a meaningful recognition result, e.g. the combined count was equal to or less than 239, and it could be found in the lookup table. When multiple proper combinations existed (e.g. S_3 can be combined either with S_2 or with S_4), spelling check could serve as the last resort to find the best possible result.

Additional pixel(s). In a few cases, a letter might contain additional pixel(s) against its pixel count in the lookup table. For example, an additional pixel occurred above 'A' in Fig 8 c). To address this problem, we relied on an approximate table lookup: when a pixel count for a segment could not be located in the lookup table, this segment would be recognised as the most likely letter.

This method does not succeed all the time, since some letters have close pixel counts (e.g., V, E and U; D, Z and S; M and W). However, sometimes, we could resort to the spelling check technique to find the correct result. For example, when multiple candidate answers were returned by the approximate method, spelling check could be used to choose the best possible solution.

4.4 Results

With all the above enhancements, our attack achieved a success rate of 92% on the sample set. To check whether our attack was generic enough, we followed the practice described in [10, 11]. We collected another set of 100 random challenges, and then run our attacks on this test set. Our attack completely broke 94 challenges in the test set. That is, a success rate of 94%. On the contrary, without image analysis, an attack program with access to the dictionary used in Scheme 1 has merely a 1/6000 chance of guessing correctly.

We did not analyse any challenges in the test set, and no additional modifications were made to our program. One arguable convenience we took advantage of is that we made sure our dictionary covered all words used in the test set. This could be avoided by using a large dictionary, although it would decrease the attack speed. However, as a security requirement, a CAPTCHA by definition should make its code and data publicly available [1], rather than relying on "security through obscurity".

Most failure cases in both the sample and the test sets were due to the same reason: the failure of vertical segmentation led to partial results such as 'S*****' and '******', which matched too many candidate words that had the same pixel sum in the dictionary. The unique exception, which was in the test set, was a failure of the spelling check to differentiate between 'P' and 'V': two alternatives were both in the dictionary.

5 Breaking Scheme 2

In Scheme 2 (*random_letters_image*), each challenge is a distorted image of a random six-letter sequence, rather than an English word. However, the challenge images in Schemes 1 and 2 share many common characteristics, such as:

- Each image is of the same dimension: 178 × 83 pixels. Only two colours are used in the image, one for background and another for foreground which is the distorted challenge text.
- Only capital letters are used. Few letters overlap or touch with each other.
- Each letter has an (almost) constant pixel count. The one-to-one mapping from a pixel count to a letter in Table 2 is still valid.

The basic attack algorithm in the previous section was also applicable to Scheme 2, and it has broken 28 out of 100 random Scheme 2 challenges we collected. However, the dictionary attack did not work here. It is possible (but expensive) to build a dictionary of 6-random-letter strings (26^6 =308,915,776 dictionary entries). However, the pixel sum matching would often return multiple candidates. Moreover, the spelling check technique was no longer applicable to differentiate letters with an identical pixel count.

To boost the success rate, we have developed a new method, largely based on the following new ideas: a "snake" segmentation algorithm, which replaced the vertical segmentation since it could do a better job of dividing an image into individual letter components, and 2) some simple geometric analysis algorithms that differentiated letters with the same pixel count.

5.1 Snake segmentation

Our snake segmentation method was inspired by the popular "snake" game, which is supported in most mobile phones. In this game, a player moves a growing snake on the screen, and tries to avoid collisions between the snake and dynamic blocks. In our algorithm, a snake is a line that separates the letters in an image. It starts at the top line of the image and ends at the bottom. The snake can move in four directions: Up, Right, Left and Down, and it can touch foreground pixels of the image but never cuts through them. Often, a snake can properly segment a challenge that the vertical segmentation fails to do.

The first step of the snake segmentation was to preprocess an image to obtain the first and last segmentation lines, as illustrated in Fig 9 (a). The first segmentation line ($X= x_{first}$) was obtained as in the vertical segmentation algorithm, and then the vertical slicing started at point (width, 0), moving leftwards to locate the last segmentation line ($X= x_{last}$).

The top and bottom edges of the image between these two segmentation lines were starting and ending lines for a snake. Since each letter occupies some width, we chose to refine the starting line by shifting 10 pixels to each segmentation lines. That is, for snakes, all possible starting points are between (x_{first}+10, 0) and (x_{last} - 10, 0).

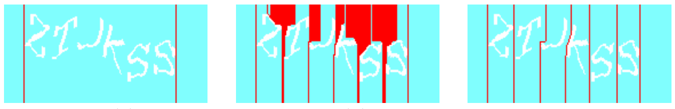


Fig 9. Snake segmentation. (a) Pre-processing: finding the first and last segmentation lines. b) Before segment finalizing. (c) After segment finalizing.

Next, the snake segmentation was started to divide the pre-processed image into segments. The following heuristics control the movement of a snake:

1. Whenever feasible, a snake moves down vertically as much as possible. That is, Down is the direction that has the highest priority.
2. A snake moves down from its starting point until it is immediately above a foreground pixel.
3. When a snake can move Left and Up only, it moves left one pixel. And then moves down as much as possible.
4. When a snake can move Right and Up only, it moves right one pixel. And then moves down as much as possible.
5. When a snake can move right and left only, it goes right. (Priority order: D > R > L > U)
6. When a snake moves left, it cannot go to any point that is to the left of a previously completed segmentation line.
7. A vertical slicing line could be a legitimate segmentation line.
8. Distance control: when a snake reaches the bottom line, it is done.
9. If a snake cannot reach the bottom, it is aborted and all its trace is deleted.
10. No matter whether or not the previous snake succeeded in reaching the bottom, the next snake starts one pixel to the right of the previous starting point.

There could be multiple snakes between two segments, see Fig 9(b), where for example the red block between 'K' and 'S' were in fact a set of snake lines that touched each other. Therefore, the last step was to finalise the segments. This process dealt with the following tasks: 1) getting rid of redundant snakes: if there was no foreground pixel in a segment, then this was an empty segment and one of its segmentation lines was redundant; and 2) when necessary, handling broken characters by merging neighbouring segments using the method discussed in the previous section. Fig 9(c) shows the finalised segments of a challenge, one for which vertical segmentation would fail to segment overlapping letters T, J and K.

5.2 Simple geometric analysis

To enhance the snake segmentation approach, we designed simple algorithms to tell apart letters with an identical pixel count by analyzing their geometric layouts.

Differentiating between 'P' and 'V'. When a segment had a pixel count of 162, it could be either 'P' or 'V'. To determine which letter it was, this segment would be first *normalised*: its left segmentation line would be adjusted to cross its left-most foreground pixel vertically and similarly for the right segmentation line. Then, a vertical line would be drawn in the middle of the normalised segment. If this middle line cut through the foreground text only once, this segment would be recognised as 'V'; otherwise, it was recognised as 'P' (see Fig 10). It was unlikely for the middle line to cut through 'V' twice, since it was rare to use a rotated 'V' in a challenge presumably due to a usability concern: it would be very difficult for people to differentiate a rotated 'V' from a distorted 'L' or 'J'.

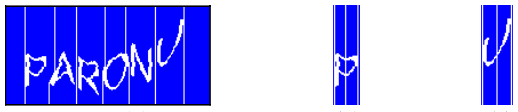


Fig 10. Recognising 'P' and 'V'. (a) A segmented challenge. (b) The 1st segment was normalised and successfully recognised as 'P'. (c) The 6th segment was normalised and successfully recognised as 'V'.

This method would not work when a 'P' or 'V' happened to have a crack in the middle of its normalised segment. However, it is trivial to address this exception: the middle line could be shifted horizontally a number of times, and each time the number of intersections it cut through the foreground would be checked. If two or more intersections occurred more often, then we are sure this segment was 'P'; otherwise it was 'V'.

Telling 'O' and 'K' apart. When a segment had a pixel count of 178, it could be either 'K' or 'O'. To determine which letter it was, a vertical line would be drawn in the middle of the segment. If this line cut through the foreground text only once, this segment would be recognised as 'K'. If this cut-through line had two intersections with the foreground, the letter could be either 'O' or 'K'. However, we observed that the distance between two intersections, denoted by *d*, was larger for 'O' than for 'K'. In our algorithm, if this distance was larger than 14 pixels (an empirical threshold), the letter was recognised as 'O'; else, it was recognised as 'K'. Fig 11 shows a segment that was normalised and then successfully recognised as ‘O’ in this way.

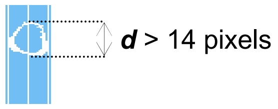


Fig11. A normalised segment was successfully recognised as ‘O’.

However, this method is not perfect. For example, if there was a break in letter ‘O’ and this break was exactly in the middle of the normalised segment, then the cut-through line would only cross the foreground once. Thus, this letter would be wrongly recognised as ‘K’. However, this kind of failure was rare in our experiment.

Differentiating between ‘L’ and ‘J’. To tell whether a segment is ‘L’ or ‘J’, the segment was first normalised. A horizontal line would then start to slice the segment horizontally from top to bottom, until it intersected the foreground text. If the intersection was closer to the left segmentation line, then the segment was recognised as ‘L’; if the intersection was closer to the right segmentation line, then the segment was recognised as ‘J’ (see Fig 12). If the intersection was exactly in the middle, it was guessed by default as ‘L’. Since this kind of scenario was rare, we did not introduce any more sophisticated methods.

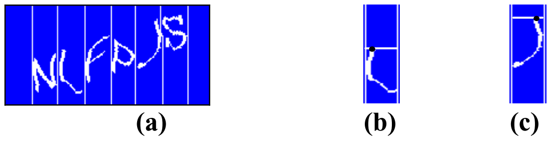


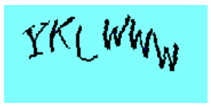
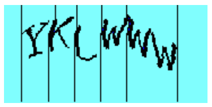
Fig 12. Recognising ‘L’ and ‘J’. (a) A segmented challenge. (b) The 2nd segment was normalised and successfully recognised as ‘L’. (c) The 5th segment was normalised and successfully recognised as ‘J’.

We do not claim that the above algorithms are generic, but they turned out to be effective in our attacks. In fact, geometric characteristics between ‘P’ and ‘V’, ‘O’ and ‘K’, and ‘L’ and ‘J’ vary so much that , when necessary, it is almost always possible to develop more sophisticated geometric analysis algorithms to differentiate them.

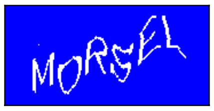
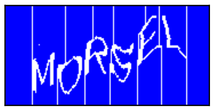
5.3 Results

Our attack program implemented the snake segmentation algorithm, geometric analysis, and the enhancements discussed in the previous section such as approximate table lookup, and the countermeasure for dealing with broken characters (but not the spelling check, which of course cannot help). The program achieved a success rate of 96% on the sample set. Another set of 100 random challenges was collected as a test set, and we achieved a success rate of 99% on this set.

There was no completely unrecognised challenge in all five failure cases. At least four or five characters were correctly recovered in each case. Failure cases were mainly due to one of two reasons: 1) failure in merging broken letters (i.e. a segment could be combined with either its preceding or following neighbour), and 2) failure of the snake segmentation to segment connected letters. Below is a failure case caused by the second reason, where ‘WW’ could not be split. We used pixel sum matching to guess non-segmented letters, but this does not always succeed. In this case, ‘QN’ was returned as the result for the last segment, since it was one of the combinations that had the same pixel sum as ‘WW’.

Original	Segmented	Recognised
		YKLWQN

When snake segmentation was applied to Scheme 1 challenges, the results were also positive. The success rate on the sample set was boosted to 99% (without using a dictionary). The only failing case was the following.

Original	Segmented	Recognised
		WORSEL

An additional pixel above 'M' caused this letter to be recognised as 'W' by the approximation table lookup method, as 'W' was larger than 'M' just by one in terms of pixel count. When a dictionary was used for the spelling check in the last step, the success rate was increased to 100%. On the other hand, the success rate on the test set was also 100% (without using a dictionary).

6 Attacks on Schemes 3 & 4

In Scheme 3 (*number_puzzle_text_image*), a challenge is a distorted image of a random number. By analysing 100 samples we collected, we made the following observations of the characteristics of Scheme 3.

- As with the previous two schemes, only two colours were used in each challenge, one for background and another for foreground, i.e., the distorted challenge text.
- Each challenge used only numbers, which consisted of 1~7 digits. The average number of digits per challenge was 2.9.
- The height of each challenge image was fixed, but its width increased proportionally to the number of digits used.

- Only seven digits were used: ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘6’, and ‘9’. ‘0’ occurred most often. All this might seem to be surprising, but a convincing explanation was suggested by a colleague [23] that ‘5’, ‘7’ and ‘8’ are omitted for sound usability reasons: 5 is very hard to tell apart from 6, 7 is written differently in different countries and often what looks like a 7 may in fact be a 1, and 8 can look like 6 or 9.
- Not many digits in a challenge overlapped or touched each other. Each challenge appeared to be vulnerable to either the vertical or the snake segmentation attack.
- A one-to-one mapping between a digit and its pixel count was established in Table 3.

Digit	Pixel Count
0	234
1	109
2	182
3	164
4	173
6	181
9	183

Table 3. A digit-pixel count lookup table for Scheme 3

Our attack was largely based on the lookup table and a segmentation algorithm. With the vertical segmentation algorithm, we achieved a success rate of 61% on the sample set, and 63% on a test set of another 100 random samples. With the snake segmentation algorithm, the success rate was boosted to 100% on both the sample and test sets.

In Scheme 4 (*user_string_image*), each challenge was a distorted image of a user-defined sequence, which had a maximum length of 15 and could include letters A-Z, a-z and numbers 0-9. The distortion method was the same as in Schemes 1-3. We observed that in this scheme, our lookup table in Table 2 was still valid. Among letters a-z, only ‘a’ and ‘y’ had the same pixel count; otherwise, each letter had a unique but constant pixel count. Each digit (from ‘0’ to ‘9’) had a unique but constant pixel count, and digits 0,1,2,3,4,6,9 had the same pixel counts as they had in Scheme 3.

It is straightforward to apply our attacks to breaking Scheme 4. However, a little more effort is needed to differentiate characters with identical pixel counts. As shown in Fig 13, there are more such characters than before. However, just as we did in the previous section, it is straightforward to develop simple geometric methods to differentiate these characters. In addition, special care might be needed to segment letters ‘i’ and ‘j’. But in general, they could be treated as broken letters.

‘J’ = ‘L’= 111	‘W’ = ‘0’ = 234
‘P’ = ‘V’ = 162	‘A’ = ‘9’ = 183
‘K’ = ‘O’ = 178	‘U’ = ‘3’ = 164
	‘X’ = ‘6’ = 181
‘F’ = ‘t’ = 133	
‘C’ = ‘e’ = 159	‘a’=‘y’ = 158
'H' = 'p' = 186	
'I' = 'v' =121	‘H’ = ‘p’ = ‘5’ = 186

Fig13. Characters with an identical pixel count in {A-Z, a-z, 0-9}.

We did not implement our attacks on Scheme 4, but we believe it is easy to achieve a similar success as in previous sections.

7 Attack Speed

We implemented our attacks in Java (little effort was spent in optimizing the run-time of code). Each attack was run ten times on the test set of each scheme on a laptop computer with a Pentium 2.8 GHz CPU and 512MB RAM, and the average speed was taken, together with the slowest one (see Table 4). The figures in the table show that our attacks were efficient: it took around 20~50 ms to break a challenge in all the schemes.

CAPTCHA	Attacks	Total samples	Success	Time (ms)	
				Avg.	worst
Scheme 1	VS + dictionary	100	94	5318	6485
	SS	100	99	3267	3875
Scheme 2	SS	100	99	4397	5031
Scheme3	SS	100	99	1709	2094

Table 4. Attack speed (“VS”: vertical segmentation; ‘SS’: snake segmentation).

The snake segmentation based attack worked more slowly with Scheme 2 (*random_letters_image*) challenges than with those from Scheme 1 (*word_image*). This was because the speed of snake segmentation is dependent on the characters in an image. For example, it is much slower to segment letters such as 'U' and 'X', since they have “valley” shapes where snakes can get trapped and then take long time to emerge from. Our observation confirmed that the Scheme 2 samples we collected happened to have used such letters more often than the Scheme 1 samples.

8 Discussion

8.1 Is *Captchaservice.org* the only victim?

Fatal design mistakes made it easy for us to break all the four visual CAPTCHA schemes provided by Captchaservice.org. Mistakes exploited by our attacks include the following.

- It was easy to separate foreground text from background with an automatic program.
- The random shearing technique as implemented was vulnerable to simple segmentation attacks.
- Constant and (almost) unique pixel counts for each character often made it feasible to recognise a character by counting the number of foreground pixels in each segment.

Many other visual CAPTCHAs used on the Internet are vulnerable to our attacks, since their designs have (or had) similar errors. We briefly discuss a few schemes as follows.

Bot Check [18] is a popular Wordpress plug-in for protecting against automated blog posts. Two versions of this tool are available, but they generate CAPTCHA challenges in the same way. Fig 14 shows some sample challenges generated by Bot Check 1.1. In this scheme, distortion mainly relies on a noisy background. However, although multiple colours are used in each challenge, the foreground is of a single colour that is distinct from the background (see Fig 14b). It was straightforward to extract the challenge text, segment it and then use the pixel count method to decode each challenge. Our attacks have broken this scheme with 100% success, although most samples we collected were resistant to the OCR program we used in our experiments.

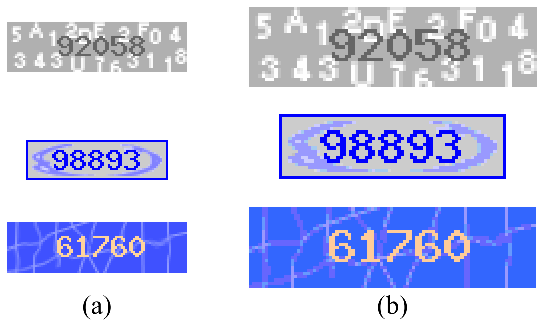


Fig14. Bot-Check 1.1 sample challenges (a) original size; (b) zoom × 2

BotBlock [20] is another visual CAPTCHA scheme for blocking spam bots from auto-filling web forms. As showed in Fig 15 (a), random letters are used in this scheme, and they appear in different places in a challenge. A sophisticated colour management method is also introduced. Backgrounds are of multiple colour blocks of random shapes, and foreground colours also occur in the background. All the samples we tested were resistant to the OCR program we used. In our experiments, we successfully extracted the challenge text (see Fig 15b) by exploiting its colour pattern -- the same colour occurs repetitively. Then, we applied the pixel count method. When necessary, geometrical analysis was also used to tell apart letters with identical pixel counts. Our attacks have successfully broken all of the 100 random samples we tested.

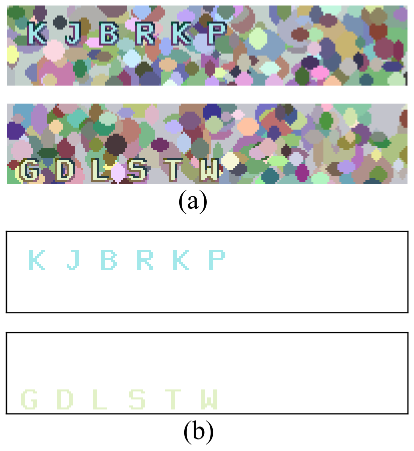


Fig15. (a) BotBlock sample challenges (b) Extracted challenge texts

HumanVerify [21] is a simple CAPTCHA scheme that claims to be used by more than 1,000 sites. Fig 16a shows some sample challenges. It appears that this scheme is also vulnerable to our attacks. In a small-scale experiment (only 10 random samples were used), our program could easily get rid of the dotted lines, and restore characters to the form as shown in Fig 16b. Then, approximate pixel counting and geometrical analysis enabled us to decode all the samples successfully.

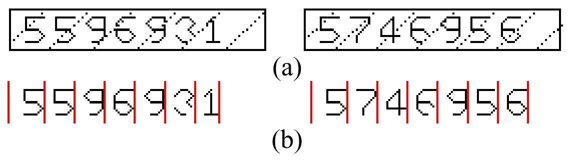


Fig16. Human Verify sample challenges

As verified in a small-scale experiment (where 10 random samples were used for each scheme), our attacks could break or aid in breaking some schemes that were listed at the PWNtcha site [9], e.g.:

- the *Clubic* scheme,
- the *Ourcolony* scheme,
- the *Scode* scheme ([19] is the vendor site)
- the *lmt.lv* scheme (still actively used at www.lmt.lv)

The *Scode* scheme is similar to Bot Check, except that multiple typefaces are used. For the last two schemes, a little additional effort would be needed to remove the grid lines in each challenge.

8.2 Lessons

The first lesson we have learnt is the following: a CAPTCHA scheme that is resistant to OCR software attacks is not necessarily secure, and it could be vulnerable to (simple) customized attacks. Without a rigorous, independent robustness evaluation, a CAPTCHA scheme might provide only a false sense of security.

Second, segmentation methods can be critical for the success rate of attacks. As shown in early sections, the snake segmentation has contributed much more to the success rate than the vertical segmentation. This echoes an observation by Chellapilla and Simard [7] that most of their failures (of breaking CAPTCHA challenges) were due to incorrect segmentations.

Scheme 4 (*user_string_image*), as discussed in Section 6, could be broken with a high success rate. This implies that the multimodal scheme, of which Scheme 4 was a part, was defeated. Therefore, another lesson is: a multimodal CAPTCHA might support better accessibility, but if any mode is weak, the entire scheme could be insecure or even useless.

We have also gained some new insights on the design of visual CAPTCHA schemes that can be of generic interest. First, for each of the visual schemes provided by Captchaservice.org, it was a useless design decision to allow the program to pick two different colours randomly for a challenge. Such a design would not introduce any advantage over a fixed two-colour scheme. Instead, it could cause serious usability problems for colour-blind people.

Interestingly, using multiple colours in a CAPTCHA scheme does not necessarily make it harder to extract the distorted text or increase the robustness of the scheme in some other way. Bot Check and BotBlock are two good examples. Another example is the well known Gimpy-r scheme: the dominant colour of distorted texts in each challenge always had the lowest intensity amongst all colours used in the challenge, and this colour (often black) never appeared in the background. This made it easy to extract the challenge text, and the colourful background was useless most of the time – rather, its negative side effect is obvious: it confuses people and decreases the usability of the scheme.

Next, although lexical information (English words in our case) made Scheme 1 (*word_image*) vulnerable to dictionary attacks, and the failure of vertical segmentation was significantly compensated for by the dictionary attacks, it is not necessarily a design mistake to make use of words. The reason is simple: one of our attacks has achieved a better success rate on the same scheme without using any dictionary. We are not convinced that it is absolutely a bad idea to make use of lexical information in CAPTCHA schemes. Instead, lexical information can improve the usability of a visual CAPTCHA scheme. For example, it might be difficult for people to recognise individual characters that were distorted too much. But when these characters occurred as part of a word in a challenge, people could easily solve the challenge using the lexical context, as suggested by Gestalt psychology [3] (i.e., humans are good at inferring whole pictures from only partial information). What really matters is how to make use of lexical information properly in CAPTCHAs. Some obvious thoughts to this end include the following: the dictionary should be large enough; when embedded into challenges, all words should be randomly picked, and, more importantly, the distortion method used should be resistant to segmentation attacks.

8.3 Defence

Simple methods that can defeat our attacks (but not necessarily other types of attacks) include the following.

- Make it hard to separate the text from the background, e.g. by using multiple colours for both foreground and background and leaving no pattern that could be used to distinguish foreground automatically, and including some foreground colours into the background and vice versa.
- Make it hard to segment each image, e.g. by having characters connected or overlapped with each other, by adding more cracks in each character, and by adding distortion such as drawing arcs above the challenge (note that arcs would be useless unless they share some or all of the foreground colours).
- Make it impossible to distinguish a character by counting its pixels. For example, all characters have the same pixel count all the time. Or a character can have very different pixel counts in different challenges (if the difference is not large enough, then probably an approximation method could be used to tell each character).

9 Conclusions

We have exploited fatal design mistakes to develop simple attacks that could break, with near 100% success, four visual CAPTCHA schemes (including one visual component of a multimodal scheme) provided by *Captchaservice.org* -- these schemes all employed sophisticated distortions, and they were effectively resistant to OCR software attacks and appeared to be secure.

It is alarming that we have also found that many other visual CAPTCHAs deployed on the Internet made similar mistakes, and thus could be effectively broken by our simple attacks. The major reasons that we suspect can explain this scale of failure are the following. Although a few pioneering efforts shed some light on how to design visual CAPTCHAs, our collective understanding of this topic is still in its infancy. There were a few design guidelines and rules of thumb scattered in the literature, but many more are yet to be identified. Neither is there a systematic method for verifying whether a CAPTCHA is indeed robust. Otherwise, the fatal design mistakes identified in this paper might have been easily avoided. Therefore, our paper calls for further research into the design of practically secure and robust CAPTCHA schemes, a relatively new but important topic, and in particular into establishing both a robustness evaluation method and a comprehensive set of design guidelines – the latter can include, for example, what should be included or avoided in CAPTCHA design, as well as what could be used.

For the same reason as speculated above, many of today's CAPTCHAs are likely only to provide a false sense of security. We expect that systematically breaking representative schemes will generate convincing evidence and, as demonstrated by this paper, establish valuable insights that will benefit the design of the next generation of robust and usable CAPTCHAs.

Acknowledgement

We are grateful to Brian Randell and Lindsay Marshall for their valuable comments and proofreading. Comments from anonymous referees also helped improve this paper. This work was partially supported by the EU funded NoE ReSIST project.

References

1. L Von Ahn, M Blum and J Langford. "Telling Humans and Computer Apart Automatically", CACM, V47, No2, 2004.
2. HS Baird, MA Moll and SY Wang. "ScatterType: A Legible but Hard-to-Segment CAPTCHA", Eighth International Conference on Document Analysis and Recognition, August 2005, pp. 935-939
3. D Bernstein, EJ Roy, TK Srull, CD Wickens. Psychology (2nd ed.). "Chap 5. Perception". Houghton Mifflin Co., 1991.
4. M Chew and HS Baird. "BaffleText: a human interactive proof". Proceedings of 10th IS&T/SPIE Document Recognition & Retrieval Conference; 2003, San Jose; CA; USA.
5. AL Coates, H S Baird and RJ Fateman. "PessimalPrint: A Reverse Turing Test", Int'l. J. on Document Analysis & Recognition, Vol. 5, pp. 158-163, 2003.
6. Casey. "Using AI to beat CAPTCHA and post comment spam", http://www.mperfect.net/aiCaptcha/, 1/30/2005.
7. K Chellapilla, and P Simard, "Using Machine Learning to Break Visual Human Interaction Proofs (HIPs)," Advances in Neural Information Processing Systems 17, Neural Information Processing Systems (NIPS), MIT Press, 2004.
8. T Converse, "CAPTCHA generation as a web service", Proc. of Second Int'l Workshop on Human Interactive Proofs (HIP'05), ed. by HS Baird and DP Lopresti, Springer-Verlag. LNCS 3517, Bethlehem, PA, USA, 2005. pp. 82-96
9. Sam Hocevar. PWNtcha - captcha decoder web site, http://sam.zoy.org/pwntcha/, accessed Jan 2007.
10. Greg Mori and Jitendra Malik. "Recognising Objects in Adversarial Clutter: Breaking a Visual CAPTCHA", IEEE Conference on Computer Vision and Pattern Recognition (CVPR'03), Vol 1, June 2003, pp.134-141.
11. Gabriel Moy, Nathan Jones, Curt Harkless and Randall Potter. "Distortion Estimation Techniques in Solving Visual CAPTCHAs", IEEE Conference on Computer Vision and Pattern Recognition (CVPR'04), Vol 2, June 2004, pp. 23-28
12. D Naccache and C Whelan. "9/11: Who alerted the CIA? (And Other Secret Secrets)", Rump session, Eurocrypt, 2004.
13. Openwall wordlists collection (free version), available at ftp://ftp.openwall.com/pub/wordlists/, accessed Aug 2006.
14. C Pope and K Kaur. "Is It Human or Computer? Defending E-Commerce with CAPTCHA", IEEE IT Professional, March 2005, pp. 43-49
15. J Yan. "Bot, Cyborg and Automated Turing Test", the Fourteenth International Workshop on Security Protocols, Cambridge, UK, Mar 2006. Also available at http://www.cs.ncl.ac.uk/research/pubs/trs/papers/970.pdf.
16. http://captchaservice.org, accessed July 2006.
17. http://www.abbyy.com/, accessed Aug 2006.
18. Bot Check 1.1, available at http://www.blueeye.us/wordpress/2005/01/08/human-check-for-wordpress-comments/, 2005. Bot-Check 1.2, available at http://blog.rajgad.com/work/software/2006-11/bot-check-12-wordpress-anti-spam-comment-plugin.html, Nov 2006.
19. James Seng, "Solutions for comments spasm", http://james.seng.cc/archives/000145.html (source code available), Oct, 2003.
20. BotBlock. http://www.chimetv.com/tv/products/botblock.shtml
21. http://www.humanverify.com
22. http://www.captcha.net/
23. L Marshall, Personal communications, 2007.
24. P Simard, R Szeliski, J Benaloh, J Couvreur and I Calinov, "Using Character Recognition and Segmentation to Tell Computers from Humans", Int'l Conference on Document Analysis and Recogntion (ICDAR), 2003.