The Robustness of Google CAPTCHAs

Ahmad S El Ahmad, Jeff Yan, Mohamad Tayara
School of Computer Science
Newcastle University, UK
{Ahmad.Salah-El-Ahmad, Jeff.Yan, Mohomad.Tayara}@ncl.ac.uk

May 8, 2011

ABSTRACT
We report a novel attack on two CAPTCHAs that have been widely deployed on the Internet, one being Google's home design and the other acquired by Google (i.e. reCAPTCHA). With a minor change, our attack program also works well on the latest ReCAPTCHA version, which uses a new defence mechanism that was unknown to us when we designed our attack. This suggests that our attack works in a fundamental level. Our attack appears to be applicable to a whole family of text CAPTCHAs that build on top of the popular segmentation-resistant mechanism of "crowding character together" for security. Next, we propose a novel framework that guides the application of our well-tested security engineering methodology for evaluating CAPTCHA robustness, and we propose a new general principle for CAPTCHA design.

Categories and Subject Descriptors
D.4.6 Security and Protection, H.1.2 User/Machine Systems.

General Terms
Security, Human factors

Keywords
CAPTCHA, robustness, segmentation attack

1. INTRODUCTION
A CAPTCHA (Completely Automated Public Turing Test to Tell Computers and Humans Apart), also known as a human interaction proof, is a program that generates and grades tests that are human solvable but intended to be beyond the capabilities of current computer programs [1]. CAPTCHA makes use of a hard, open problem in AI, and is now a standard technology to defend against undesirable or malicious computer bot programs. The most widely deployed CAPTCHAs are text-based schemes, which require a user to recognize distorted characters, a task that state-of-the-art AI programs supposedly cannot do.

CAPTCHAs' robustness is the strength of their resistance to AI programs written to automatically solve CAPTCHA tests. It is well known that the robustness of a text CAPTCHA should rely on the difficulty of finding where each character is rather than what it is. The rationale is the following. Computers perform better than humans in recognizing individual characters, even under severe distortion [6]. However, locating individual characters in the right order (i.e. segmentation) is in general a computationally expensive and combinatorially hard problem for computers. Therefore, a text CAPTCHA should be segmentation-resistant; if a scheme is vulnerable to a segmentation attack, then it is effectively broken. A commonly accepted goal for CAPTCHA design is that automated attacks should not be more than 0.01% successful but that the human success rate should be at least 90%.

Various segmentation-resistant mechanisms have been proposed (some representative ones are shown in Figure 1), but many of them were broken, including those developed by major companies such as Microsoft, Yahoo and Megaupload [3, 10]. A segmentation-resistant mechanism known as "crowding character together" or CCT, initially adopted by Google to let adjacent characters touch or overlap with each other at a random intersection (see Figure 1 for illustrations), was shown to be more resistant to known attacks [3]. This mechanism has been used by Google until now, and it has gained wide popularity and been in use by a large number of different text CAPTCHAs.

Figure 1. CAPTCHAs designed by Microsoft, Yahoo, Megauplad and Google (clock wise), each using a different segmentation-resistant mechanism.

This paper first answers an open problem that has intrigued the CAPTCHA community for years: Is the CCT mechanism vulnerable to novel attacks? Our analysis of a Google's CAPTCHA, as currently deployed by Gmail, Blogger.com and BlogSpot, suggests that a simple but novel attack can break its CCT mechanism. This is to date the most effective attack against this scheme (For convenience, we will refer to this CAPTCHA as the Google scheme in this paper).

Our attack is also applicable to ReCAPTCHA, a scheme that is widely used by millions of users of Facebook, Twitter and other Internet services on a daily basis (used by over 100,000 websites), and that was recently acquired by Google. Our attack was implemented for a ReCAPTCHA version that was active in May 2010. To our surprise, our attack, without any significant change, also works well on the latest ReCAPTCHA version, which uses a new defence mechanism that was unknown to us when we designed our attack. This suggests that our attack works in a fundamental level.

Our attack exploits shape patterns found in individual characters (such as the "loop" shape in characters "a" and "p", and the cross shape in characters 'f' and 't'), as well as connection patterns of adjacent characters. The key insight is that these features and patterns largely stay invariant under various distortions and transformations applied by the CAPTCHA generators, and can be exploited to segment connected characters.

Given the CCT mechanism's popularity, the impact of our attack is beyond the Google scheme and ReCAPTCHA. We urge the

designers of other captcha schemes that rely on the CCT mechanism to carefully check how vulnerable their designs are to our attack or its variants.

In the recent years, we have been working to establish a novel security engineering approach to the evaluation of CAPTCHA robustness through a series of work, including [3, 4, 7, 10] and this paper. In contrast to a parallel approach developed primarily in the computer vision, document analysis and pattern recognition communities, where sophisticated objection recognition algorithms are often the design goal, our approach applies adversarial thinking skills searching for and exploiting vulnerabilities hidden in CAPTCHAs.

In this paper, we will summarise our approach, and for the first time provide a detailed framework that classifies CAPTCHA vulnerabilities into major categories. This framework provides key insights on security vulnerabilities that text CAPTCHAs should account for. It can be used to examine a CAPTCHA's robustness, as well as to guide the design of next generation CAPTCHAs. We will also summarise general principles for the design of robust text CAPTCHAs. We not only revisit established principles, but also propose a new one.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 presents our attack on the Google scheme, and Section 4 shows that a variant of our attack works on ReCAPTCHA. Section 5 discusses further applicability of our attack and its defence. Section 6 introduces our framework for understanding Captcha vulnerabilities, and Section 7 focuses on general principles for CAPTCHA design.

2. RELATED WORK

CAPTCHA has been an active field in the research communities. Due to space limit, we only review the literature that is most relevant to this paper. Kurt's thesis [18] provided a good review of CAPTCHA research.

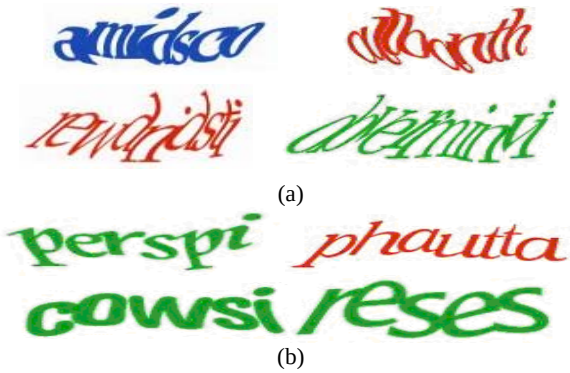
Chellapillas' et al [5] attacked a number of early CAPTCHAs using machine learning algorithms, and they achieved 4.89% success on an early version of Google's CAPTCHA (around year 2004. In 2007, we developed a technique that exploited gaps (whit e space) between characters in Google's CAPTCHA at the time, and our attack achieved 12% success [3]. It was reported in [14] that some spammers succeeded in breaking the Google CAPTCHA using two compromised zombie hosts, with each host using a variation of their attack. This attack claimed a success rate of 20%, yet no technical details have been revealed. An attack on ReCAPTCHA using a combination of image processing and OCR techniques was reported with a success rate of 17.5% [13]. At DEFCON'18, Houck presented another attack on ReCAPTCHA; using both character segmentation and character template matching technique, his attack achieved 10% success on an early version of ReCAPTCHA and 31% success on a more recent version [8].

Other notable attacks on (image recognition) CAPTCHAs include [3, 12].

3. A SEGMENTATION ATTACK on the GOOGLE SCHEME

In this section, we first review key features of the Google scheme, and then present a novel segmentation attack.

Google have tweaked their CAPTCHA from time to time in order to improve its robustness. Figure 2(a) shows a version of the Google scheme that is not user-friendly. For example, it is difficult for users to tell between "d" and "cl", "ri" and "n", "rn" and "m", and "w" and "vv". We did not choose to work on this version of Google CAPTCHA, because even human cannot decode them properly with a high accuracy. For such captchas, users' complaint will soon become a major issue or even push them offline, just as we have seen it again and again. Figure 2(b) shows another version of Google CAPTCHA, which still adopts the CCT mechanism but is easier for humans to solve than the other version, and therefore has been deployed most of the time in our experience. We have chosen to develop our attack based on the second version.



(a)

(b)

Figure 2. Google CAPTCHA. (a) a user-unfriendly version; (b) a usable version.

We studied a hundred samples that was randomly collected from the Internet[1], and observed the following features.

- Each challenge uses only two colours: Red, Green, or Blue for the challenge text, and White for the background.
- CCT is the main segmentation-resistant mechanism so that characters connect or touch with one another horizontally.
- Global warping is applied to add randomized distortion.
- The thickness of characters varies much; even different portions of the same character differ in thickness with each other. This is a powerful feature for defending against potential segmentation attacks.
- Random text strings are used; a text string's length varies between 5 and 8 characters; only lower-case letters are used.
- Multiple font typefaces and styles (such as bold, italics and regular) are used.

The key insight behind our attack is that although the Google CAPTCHA uses different font typefaces and styles, and uses heavy distortions, shape patterns in some characters remain invariant, e.g. the "loop" in characters "a" and "p", and the cross shape in 'f' and 't'. Our main idea is to first detect distinctive shape patterns, identify their associated characters, and then cut out these characters. It turns out that once these detectable

[1] All samples we used for this paper were randomly collected from Google email registration page and Blogger.com, where the CAPTCHA was used whenever one attempted to create an account or post a comment.

characters are cut out, most often a whole challenge text is also properly segmented already.

Our attack includes the following sequential steps:

- Pre-processing – A set of standard techniques is applied to prepare each challenge image.
- Pattern based detection of characters – A set of algorithms is used to locate characters with their detectable shape patterns.
- Character segmentation – A set of rules and heuristics is used to segment detected characters.

3.1 Preprocessing

We first perform image up-sampling, which enlarges the image, increases its pixel details, and thus smoothes the embedded text. Then, we convert the up-sampled image into a black-and-white image. This binarising process is done via the standard thresholding method: all the pixels with a color value above a heuristically predetermined threshold is converted to black and those bellow it to white. Next, we apply Zhangs' algorithm [16] to thin the image. Thinning is the process of identifying a binary image's skeleton. Figure 3 shows the output of our preprocessing on a sample image.

The main advantages of using thinning in our attack are: 1)Thinning normalizes the character thickness, which varied very much in a random way, to a uniformed one-pixel thickness. This greatly simplifies our attack, as it does not have to account for different character thicknesses. 2) It is much faster to process a thinned image than an un-thinned one, as the former contains far fewer pixels that matter. Clearly thinning does not segment characters – connected characters stay connected after thinning.

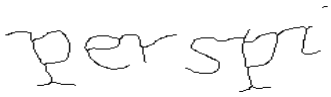


Figure 3. After preprocessing (the image is not in its actual size due to space concerns; the original image is in Figure 2(b))

3.2 Pattern Based Characters Detection

In this step, we aim to detect categories of characters in a thinned image using shape patterns found in their generic shape. We identified four shape patterns and each belongs to a category of characters as follows.

- Dot shape: "i" and "j".
- Loop shape: "a", "b", "d", "e", "g", "o", "p", "q".
- Cross shape: "t" and "f".
- S shape or "S Vertical Histogram": Characters that contain three vertically juxtaposed lines, the character "s"

Detecting Characters that contains a Dot Shape. We first use "Color Filling" segmentation on the foreground components (on the thinned version of an image). A foreground component contains a single character, a group of connected characters, or a part of a character, and has a black color. "Color Filling" Segmentation or CFS is effectively like using a distinct color to flood each component, and it could be used to segment against any color, so we call it "Color Filling" segmentation, more about CFS in [3] . In Figure 4, each foreground component is segmented by CFS and is identified by its unique color.

Figure 4. Segmentation of connecting component using CFS. Each component is indentified by a different color.

Characters such as "i" and "j" consist of a dot and a body part underneath the dot. We detect both parts as follows:

a) To detect the dot part: We use its relatively small pixel count (i.e., pixel count is the number of pixels in an component).

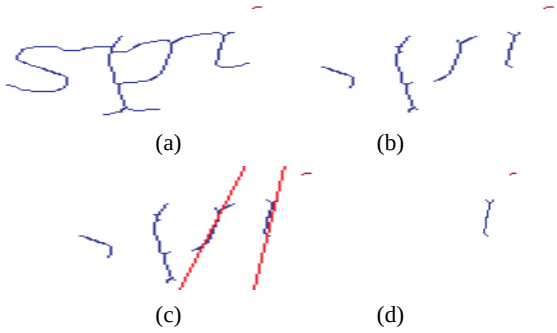
b) To detect the body part: we apply a series of steps as follows:

First, using the position of the dot, we locate all foreground components underneath it. In Figure 5 (a), only the component "spi" is positioned underneath the dot.

Second, we use a modified version of CFS (flood up and down only) to ignore all parts of the components with a horizontal orientation. Figure 5 (b) shows the remaining components having a vertical orientation.

Third, for each of the remaining components, we calculate the equation of a line representing its orientation (components far from the dot are ignored). In this case, only two lines are plotted as shown in figure 5 (c).

Finally, the component with the line path closest to the dot component is considered as the body part of either "i" or "j". Figure 5 (d) shows the only remaining component; in this case the body part of "i".



(a) (b)

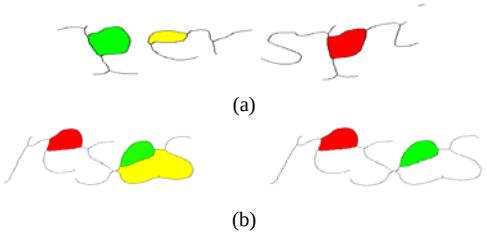
(c) (d)

Figure 5. Detecting characters with a dot shape: (a) a connected component located under the dot .(b) components with a vertical orientation. (c) Plotting the line equation for the remaining thinned components underneath the dot. (c) The body part of "i".

Detecting Characters with a Loop Component. We adopted a loop detection method that we used in [3], and which involves two steps as follows:

First, CFS is applied to the background color (i.e., white) of the image. As shown in Figure 6 (a), background components are now segmented and identified by different colors. Second, the above step return two types of loop components, "character loops" and "connection loops". Connection loops are created as a result of the crowding of characters together. Characteristics of connection loops are: 1) a relatively small pixel count, or 2) a

relatively large pixel count if vertically overlapping and in close proximity with character loop(s). We developed heuristics based on the pixel count and the relative position of loops to detect and remove connection loops. Figure 6 (b) shows a different example containing a "connection loop" before and after removal.



(a)

(b)

Figure 6. Detection of characters with a loop shape. (a) CFS on the background color is used for loop detection. (b) An example of a connection loop before and after removal.

Detecting Characters with a Cross. A unique characteristic of a cross shape is having four sides; upper, lower, left and right sides. We observed that drawing an imaginary box around the cross shape must intersect with the box once from each side, with each intersection representing one of the cross shape four sides.

We detect the cross as follows. a) We traverse the image using the imaginary box, and if each of the four sides of the box intersects with one and only one foreground colored pixel, then the box position is labeled as a possible cross shape component. After that, we shift the box position and continue searching for other cross shapes, until the entire image is traversed. b) We filter through all the possible cross shapes, and we keep only those satisfying these conditions. First, the position of the cross shape is in the upper side of its foreground component. Second, all the foreground pixels covered by the box area are connected with each other (we used CFS to verify this condition), this condition is needed as all the pixels in a valid cross shape are connected with each other Finally, the cross shape must not overlap vertically with a loop shape. In Figure 7, the red box indicates the imaginary box and thus the location of a cross shape.

Figure 7. Detecting characters with a cross shape (the detected cross shapes are highlighted by a red box).

S Vertical Histogram. The unique shape characteristic of the character "s" is that it contains three vertically overlapping strokes in its shape. We detect it as follows. First, we map the image against a vertical histogram that represents the total number of foreground pixels in each column. Then we ignore all parts of the histogram that intersects with other character shapes (this is done to insure no false detection of characters having three vertically overlapping strokes, such as "a" or "e"). Second, we search the histogram for consecutive occurrence of columns with the value of three or more pixels in each column; we call such occurrence of columns as the "s-span". Finally, if an "s-span" has a width larger that 25 pixels (a threshold for the character "s" minimum width; i.e., the component under analysis has a width large enough to contain an "s"), then we use the s-span's left-most and right-most columns as a reference to draw a bounding box around the characters "s". Figure 8 shows the histogram (magnified by a factor of 4) and the "s" character bounding box.

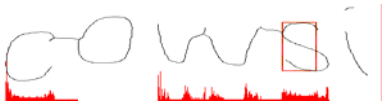


Figure 8. S Vertical Histogram. Identification of the character "s" location, as highlighted by a bounding box.

3.3 Segmentation

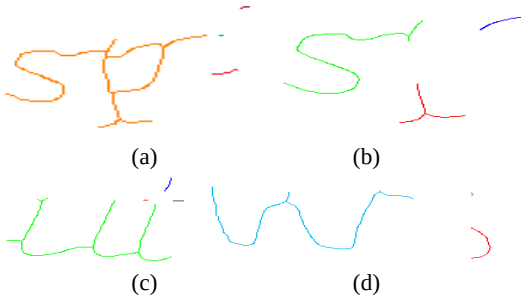
In this step, we cut out characters that have a shape pattern detected in the previous step. We use the examples used in the previous section to show how we separate 'i' from 'sp', and how to split 'sp', 'ut' and 'ws' – four examples illustrate how to segment a character with a dot, a character with a loop, a character with a cross, and a character with "s" shape, respectively.

We first convert the detected shape pattern's color to white (i.e. the image background color). This effectively hides the detected shape, and breaks connected characters into separate components, as shown in Figure 9. Note: for a character with dot, the detected shape includes the dot, and the vertical part of the body.

All visible components in Figure 9 can be classified into two types. The first type belongs to only one character, and we call them *private* components. For example, in Figure 9(a), the component in green color and the component in red were previously both connected to the body of the character "i". They belong to this character only, and therefore are private components. The second type of components occurs as the result of connected characters; these components do not belong to a single character alone, and we call them *shared* components. In Figure 9 (a), the component in brown color is a shared component, since it consists of a stroke that was previously connecting characters 'p' and 'i'.

Similarly, in Figure 9(b), private components are the blue one and the red one, and the green component is a shared one. In Figure 9(c), the green component is a shared one, and all others are private components. In Figure 9(d), the blue component is a shared one, and all others are private components.

It is simple to automatically differentiate between private and shared components: a shared component connects with other characters, and therefore has a much larger pixel count than a private component does.



(a) (b)

(c) (d)

Figure 9. Locating shared and private components.

As such, the task for segmenting a detected character becomes identifying where to cut in shared components. The properly cut shared components, a detected shape pattern and its associated private components will form a complete character.

Identifying Cutting Points. The location of a cutting point on a shared component is dependent on the nature of the character, which the component connects to.

For a shared component that connects with a character with a dot shape, the cutting point is close to the character in terms of horizontal distance. The reason is that such a character has a small width, and if we cut far away from the character, we will likely destroy its connect character(s). The cutting point we choose will make sure that we preserve both the dot character and its adjacent characters. The identification of cutting points for shared components that connect with a character with a cross pattern is similar, and for the same reason.

For a shared component that connects with a character with a loop, the cutting point is farther way from the character in terms of horizontal distance. The reason is this: a loop shape typically is inside a character; if we cut too close to the character, we will destroy it. Similarly, for a shared component that connects with an 's' shape, we cut at a point that is far away from the character with 's" shape.

Figure 10 gives an example of identifying the cutting point. Since the shared component, connecting characters "u" and "t", is positioned to the left of the cross shape and starts from the lower side of the cross shape, the cutting point is estimated at 15 pixels in horizontal distance to the left of the cross shape. The arrow indicates a distance of 15 pixels, and a red circle highlights the cutting point.

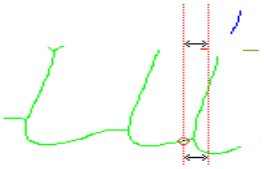


Figure 10. Locating a cutting point in a shared component.

Cutting. Cutting points are identified in a thinned image, but our real cutting is done in the image's un-thinned version. We could do the segmentation in the thinned image. However, cutting the non-thin version has advantages. First, we can reuse the rate of recognizing individual characters in Google CAPTCHA reported in the literature for estimating our overall success (segmentation and then recognition) of breaking the Google scheme. It is useful future work to check whether recognizing thinned individual characters works better than recognizing un-thinned ones, but not important for this paper. Second, the un-thinned version preserves original character shapes, which as discussed later allow further improvements to our attack.

We first copy cutting points from a thinned image to its un-thinned version. This is done, as illustrated in Figure 11, by superimposing two images, since they have the exact same width and height. Then, we draw an imaginary box (6x15 pixels in dimension, illustrated in red in Figure 11) around the cutting point, and within this box, we try to find the shortest path that can cut through a character stroke. If such path exists, then we cut through it, else we cut vertically at the location of the cutting point.

The shortest cutting path exists in the case shown in Figure 11, and is identified as follows. The green color represents a set of points S1, located in the upper side of the imaginary box. The blue color represents a set of points S2, located in the lower side of the box. To find the shortest path that can cut through the character stroke shared by "u" and "t", we compute the distance between every point in S1 to every point in S2. The points with the shortest distance are then used to cut through the character stroke. In Figure 11, both of the upper and the lowers sides of the imaginary box extended outside the area of the character stroke. But, in some cases, the upper side, the lower side, or both upper and lower sides of the imaginary box remains inside the character stroke. In such cases, the character stroke is cut vertically at the position of the cutting point.

Figure 12 shows the output of cutting the shared components in the non-thin version of the characters, where each segmented character is highlighted with a distinct color.

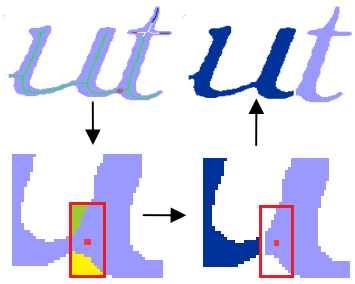


Figure 11. An example of segmenting a shared components.

3.4 Tuning

The order of character detection and segmentation is about which shape character is to be detected and segmented first (when multiple options exists), and this has an impact on our attack's success rate. The optimal order we found is to first process (detect and segment) characters with a dot, then characters with a loop, next characters with a cross, and finally "s"-shape characters.

This is mainly a decreasing ranking order in terms of false positive rates introduced by each method. For instance, the dot shape has a unique shape, and its detection method has only 1% false positive. As a result, its order was first.

On the other hand, the arrangement of characters and their connection patterns resembled character shapes in some cases. For example, we found that some of the connection patterns between characters resembled a cross shape, leading to a false detection of the cross shape. For example, the connection pattern between the characters "e" and "s" in Figure 12. In addition, horizontally overlapping italic font in connected characters could be confused with the character "s". For this, we decided to use the loop method second in order, as the segmentation of loop character lowers the chance of false "cross" and "s" shape patterns.

Since the "s" detection method is restricted to the analysis of wide characters only, we decided to use it last after the cross detection method, thus lowering the chances of confusing horizontally overlapping connected characters with the "s" shape.

Among our segmentation results in Figure 12, in "perspi", the connection between "pi" was segmented first using the dot segmentation method, followed by the segmentation of the connection between "er" and "sp" using the loop detection/segmentation algorithm; in "phautta", the connection

between “ha”, “au” and “ta” was segmented first using the loop segmentation method as no dots were detected in this case and the connection between “ut” was segmented using the cross segmentation method; in “cowsi”, the connection between “co” was segmented using the loop segmentation method first, followed by the segmentation of the connection between “ws”, and finally, in “reses”, only the loop segmentation method was used.

Figure 12. Segmentation results of the Google scheme: each segmented character is highlighted with a distinct color.

Up-sampling has an impact on both our attack’s success rate and speed. We tested with different up-sampling ratios such as 1, 2, 3 and 4. The higher the up-sampling ratio is, the higher success rate our segmentation attack can achieve, and the slower the attack is. The explanation is simple: up-sampling enlarges an image, and therefore it slows down the attack; up-sampling also smoothes characters and their connection areas, reducing segmentation errors. We identify that the optimal up-sampling ratio is 3, which achieved a reasonably good balance between the attack success and speed. Measurements reported in this paper are based on this configuration.

3.5 Attack Success and Speed

Our attack achieved a success rate of 68% on a sample-set of 100 challenges. Following a common practice in the areas of computer vision and machine learning, we tested our attack on 400 independent samples from a test-set and achieved a success rate of 62%[2]. We did not use any of the test-set samples in our attack design, as the test-set aims to generalize our attack on independent samples. That is the attack is generic enough to all challenges generated by this version of Google CAPTCHA. Given that the state-of-the-art can achieve a success rate of 95% in recognizing individual segmented characters [6], and an average number of characters in Google CAPTCHA of 5.5 characters, our attack implies that it could lead to an overall (segmentation and then recognition) success rate of 46.75% (62 * 0.95^5.5) for breaking this Google CAPTCHA.

We implemented our attack using Java, and tested it on a desktop computer with a 2.4 GHz Intel Core 4 CPU and 4 GB RAM. We ran the attack 10 times on both the sample and test sets to compute its speed and on average our attack took 7 seconds to segment a challenge.

3.6 Further Enhancements

It is important to note that when more connection patterns between adjacent characters are considered, we can significantly improve our attack’s success. We designed an algorithm to detect an interesting connection pattern between characters such as “cy”, “oo”, “bc” and “bd”. This connection pattern is called “double v”, as it (shown in Figure 13(a)) resembles a “v” shape in the upper side, and a reversed “v” shape in the lower side.

[2] Our sample set was collected in June 2009, and the test set in August 2010; both dates were random choices.

Our algorithm work as follows: First, the contour of suspicious components (i.e., components with a width that could accommodate more than one character) is mapped to a coordinate plane. We analyze the plane points from left to right. To detect a “v” shape, we search for consecutive points that have an increasing Y value and then a decreasing Y value – The higher the value of Y, the lower the point position in the image. To detect a “reverse v” shape, we search for consecutive points that have a decreasing Y value and then an increasing Y value. Then, we compare the position of the “v” and the “reversed v” shapes, and a double v pattern is detected if any of them overlaps vertically.

To segment the “double v” connection, we simply cut from the lowest point in the “v” shape, to the highest point in the “reversed v” shape. Figure 13(b) shows a segmentation result.

(a) (b)

Figure 13. Google CAPTCHA: (a) A “double v” connection pattern. (b) Segmenting result.

With this “double v” enhancement, our attack has achieved a segmentation success rate of 74% in the sample-set, and 69% on the test-set. This is so far the most successful attack on a usable version of Google CAPTCHA.

4. THE ROBUSTNESS OF RECAPTCHA

ReCAPTCHA is similar to Google CAPTCHA since both the schemes deploy the “crowding characters” mechanism and both lack defenses against attacks exploiting character shape patterns and connection patterns. In addition to its functionality as a human verification tool, ReCAPTCHA is utilized as a crowd-sourcing system for digitizing books, i.e., a text “labeling” tool. As shown in Figure 14, a ReCAPTCHA challenge employs two text strings where the answer to one of those is known to the server and thus functions as a CAPTCHA, whereas the answer to the second one is unknown and it is used for labeling functionality. The other crucial difference in ReCAPTCHA can be found in its text challenges in which, unlike Google’s, its character set includes numbers. Moreover, some of its challenge strings are dictionary words.

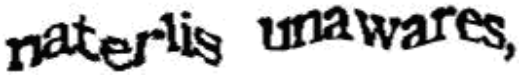


Figure 14. ReCAPTCHA: a challenge sample.

To show that our attack on Google CAPTCHA is applicable to other schemes, we developed a variant of the attack for ReCAPTCHA, and it works as follows.

4.1 Preprocessing

In this step we first divide a challenge into two images, each containing a challenge string. This is done by mapping the challenge against a vertical histogram representing the total number of black pixels in each column. Then, we search the histogram for a column satisfying two conditions: first, it contains no black pixels and, second, its position along the x-axis is the closest to the mid value of the image width. We cut through this column to divide the challenge into two images. Next, each of the two images is up-sampled by a factor of three and then binarised.

Unlike the Google scheme (where its text challenges are automatically generated and don't contain characters with deformalities), in ReCAPTCHA the characters were less smooth and often contained missing parts. This quality degradation is likely introduced by the scanning process, as ReCAPTCHA makes use of text materials from old books (that cannot be recognized by OCR systems). Missing parts of characters make thinning of little use, since many irregular strokes would be created in a thinned image.

4.2 Pattern Based Detection of Characters

In this step we aim at locating individual characters using shape patterns found in characters and their connections.

We classified three categories of characters in ReCAPTCHA as follows:

- Characters containing a dot
- Characters that are detectable by our "S Vertical Histogram" algorithm, such as "S", "3", "e" and "E";
- Characters containing a loop, e.g. "o", "b", "0", "6" and "8"

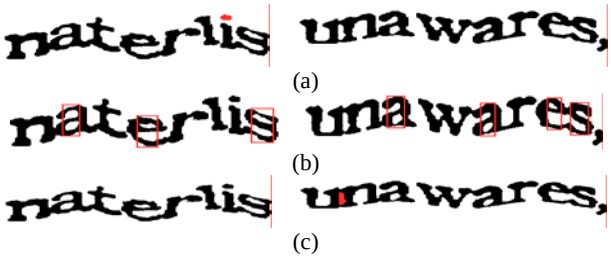
Cross detection is not needed for this attack.

Detecting Characters with Dots. We first apply CFS on the foreground components (black color). A component is considered to be a dot if it is located in the upper middle part of the image and has a small pixel count. Unlike Google's version of this algorithm, we do not attempt to detect the body part of dot characters, as it is often directly under the dot, and thus it is segmented in a similar manner to other shape characters. Figure 16(a) shows an example.

Detecting Characters using the "S Vertical Histogram". We use an algorithm similar to the one used in Google attack but, since no thinning was applied on ReCAPTCHA, the algorithm was modified to function with non-thin characters. Instead of counting the number of foreground pixels in each column, the algorithm search for, and count, a pattern found in consecutive pixels of each column. The pattern is a pixel with a background color having its upper neighbor pixel with a foreground color. Moreover, we applied fewer constraints for this algorithm. For example, the algorithm can analyze and detect components that contain loop(s). The idea of using this algorithm against "loop" characters is aimed at addressing the limitation of our loop detection algorithm in which, even if some "loop" characters contain broken loop(s) and are not detectable by our loop detection algorithm, they still could be detected by the "S vertical histogram". This enables the algorithm to detect characters such as "a", "e", "g", "z", "B", "E", "2", "3", "5", "6", "8", and "9". For example, in Figure 15 (b), in addition to the detection of the character "s", the characters "a", and "e" were also detected.

Detecting Characters with Loops. We use the same loop detection method we used for our attack on the Google scheme. For example, it detects characters such as "b", "p", "q", "0", "A", "O", and "4". In addition, we added a method to analyze loops resembling the shape of a connection pattern between adjacent characters. For instance, we considered loop shapes with a small width compared to its height to be connection pattern loops. Figure 15 (c) shows an example of a connection pattern loop between the first character "u" and the second character "n", which is highlighted in red.

Note: the order of applying these three detection algorithms and their associated segmentation operation does not matter, as explained in Appendix.



(a)

(b)

(c)

Figure 15. ReCAPTCHA shapes detection: (a) Detecting "i" using the dot shape. (b) Detecting "a", "e", and "s" using the "S vertical histogram". (c) Detecting a connection loop.

4.3 Segmentation

4.3.1 Segmenting Detected Characters

A vertical segmentation method is applied to segment detected characters. This process of vertical segmentation starts by mapping the image to a histogram that represents the number of foreground pixels per column in the image. Using the position of every detected character shape pattern as a reference, (we heuristically picked 8 columns from each side of the shape patterns in the search as it produced a better segmentation output in comparison to other values). Columns with the lowest number of pixels indicate the position of the left or the right side of a character. For detected characters, a chunk is located between the left and the right vertical lines of a detected shape and contains one character. Figure 16 (a) shows an example of vertically segmenting the character "i", in (b) the characters "a", "a", "e" and "s". In the case of connection pattern loops, we simply cut vertically through their middle. Figure 16 (d) shows such an example of segmenting a connection pattern loop between the first character "u" and the second character "n".

4.3.2 Segmenting Undetected Characters

If the horizontal distance, denoted by *d*, between the boundaries of two detected characters is large enough, we know there are undetected characters between the detected characters. These undetected characters define a new chunk of a width *d.*

We also calculate average width of detected characters in a word, denoted by *w*. By comparing *d* and *w*, we can guess with a high probability how many undetected characters there are between the two detected characters, and segment them properly. We use the following heuristics:

If *d* is larger than or equals to *w* but smaller than or equal to 2*w*, then the chunk of undetected characters is analyzed using CFS. If CFS algorithm returns two foreground components with a relatively large pixel count, then we split the chunk into two characters. This method is directed at segmenting overlapping small characters with no shape patterns, such as the characters "r" and "l" in Figure 16 (e).

If *d* is larger than 2*w*, then the bounding box of the chunk contents is analyzed. If the bounding box is double or more of its width, then we split the contents evenly into two chunks, each containing one character. This method exploits the overall shape of connected characters and it is directed at connected characters with no shape patterns, such as the connection of "th".

Otherwise, the chunk is assumed to contain only one character.

Drawbacks of this approach are the troublemaker characters with a large width, such as the characters “m” and “w”, and chunks containing a combination of connected characters with a small width, such as “ll”, “rr”, “rt”, “ln”. Figure 16 (e) and (f) shows chunks containing un-detected character(s) (highlighted by a rectangle under them). In this case, only one chunk has a width larger than the average width and contains two large components (“r” and “l”) as shows in Figure 16 (e), where as the characters “n” and “t” have a width smaller than the average. Figure 16 (g) and (h) show the final segmentation output.

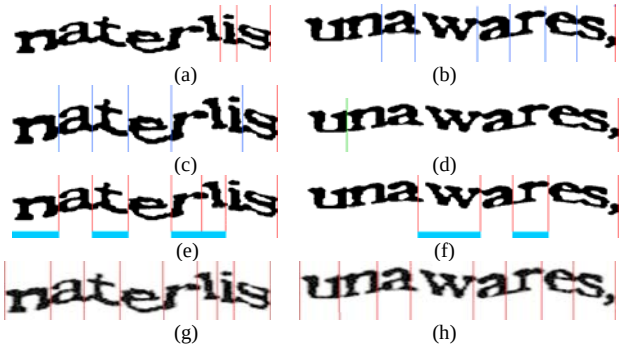


Figure 16. The final output of ReCAPTCHA segmentation attack: (a) “dot” segmentation., (b) “S vertical histogram”. (c) “S vertical histogram”. (d) “loop” connection pattern. (e) and (f) chunks segmentation. (g) and (h) shows the final output.

4.4 Results

To evaluate our attack, we downloaded 100 random ReCAPTCHA samples as a sample set, and another 100 random samples as a test set. We manually identified via online tests which are known (verification) and the unknown words in each sample. Our attack successfully segmented 53 known words in the sample set, and 46 known words in the test set. This indicates that our attack can effectively achieve a segmentation success rate of at least 46%.

We ran the attack 10 times on both sample sets and on average our attack took 0.85 seconds to segment a challenge consisting of two words. The significant difference in time between the attack on Google and that on ReCAPTCHA is due to the use of less pattern detection and, a much simpler segmentation algorithm in ReCAPTCHA attack.

Given that the state-of-the-art can achieve a success rate of 95% in recognizing heavily distorted individual characters [6], and with the average number of characters in ReCAPTCHA of 6.41 characters per word, and the segmentation success rate of 46% on the known words, our attack implies that it could lead to an overall (segmentation and then recognition) success rate of 33% (46*0.95^6.41).

Many of the cases which we treated as a failed segmentation in the above attack had, in fact, achieved a good partial segmentation. We counted 30 failures in the sample-set (12 known words) and 28 failures in the test-set (9 known words) with only one un-segmented chunk containing two or three characters. Therefore, a dictionary attack would complement well with our segmentation attack; a partial segmentation will lead to a partial recognition result, which can be used to derive a string pattern such as ‘**xxxx’ (* represents an unrecognized character). With the aid of a dictionary, the overall success would be significantly better than the rate we estimated above. For example, we tested the chunks returned by our segmentation algorithm in Figure 17 using the Tesseract OCR engine, the OCR returned the following output for each chunk respectively: “a” ,“n” , “o”, “ch”, “e” and “r”. Then we crossed referenced the output with a dictionary list of words, only one word that starts with “ano” and ends with “er” was matched, that is “another”.

With the above dictionary attack as an enhancement, our theoretical estimate of the attack success can be boosted by 9%. This implies that theoretically an attacker can break this reCAPTCHA about 42% of the time. We tested our attack, with the Tesseract OCR engine being used for individual character recognition, and the success rate for the attack is about 24.7%. The gap between the empirical and theoretical estimate is caused by the quality of OCR engine, which achieved only 84.6% success for character recognition.

Figure 17. An example of a partial segmentation. The adjacent characters “t” and “h” are not segmented.

4.5 Implications

Our attack on reCAPTCHA reveals the weaknesses of the scheme, e.g., we found a pattern that correlate between the horizontal spacing of connected characters and their number; this allowed us to exploit ReCAPTCHA using simple tools such as even cut. In addition, we found a pattern in the shape of connected characters: the bounding box of such connection shape helped us to differentiate wide horizontal spacing characters such as “m”, and the connection of small horizontal spacing characters such “th”.

Our attack also reveals insights on the defcon attack, as summarized in Appendix. First, the “wave distortion” removal in [8] is not a necessary step. Second, the core weakness of reCAPTCHA is still in its segmentation resistance mechanism.

5. Discussions

5.1 Attack Applicability to other CAPTCHAs

In early May 2011, we noticed that ReCAPTCHA has rolled out a new protection for known words to increase their robustness to attacks, see Figure 18 (a), in which the heavily distorted are known words. We downloaded 100 random samples from the Internet, and tested the attack we developed in the previous section. We only tuned the loop detection algorithm to reject loop shapes that are small in size and to reject loops that are too wide - the new distortion mechanism often creates false loops with a small pixel count or with a large ratio of width to height. All other parts of the source code stayed unchanged. The results are amazing: our program achieved a segmentation success rate of 29% on known words; that is, 29 out of 100 samples were completely segmented. Figure 18 (b) shows example output of our attack. On average, it took 0.6 seconds for our computer to segment a known word.

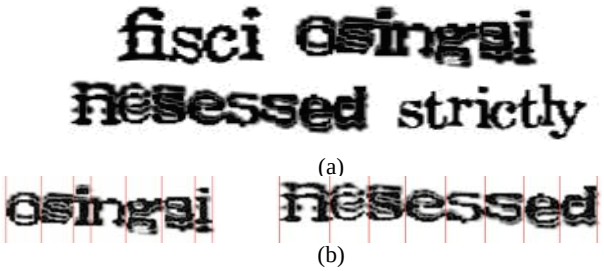


Figure 18. ReCAPTCHA as of early May 2011. (a) two sample challenges (b) completely segmented known words.

Other major CCT-based CAPTCHAs such as those from Yahoo, E-Bay, MySpace and Baidu (see Figure 19) also appear to be vulnerable to our attack or its variants, for two reasons. First, all of them are easy to pre-process, and second, all of them contain exploitable shape patterns. We suspect such an attack will successfully segment Baidu, E-Bay, MySpace, and Yahoo, but with a decreasing order for the success rate. We urge these companies to evaluate how vulnerable their designs are to our attack or its variants.

Figure 19. Other CCT-based CAPTCHAs from Yahoo, MySpace, E-Bay and Baidu (clock-wise).

5.2 Defense

Perhaps the simplest defense against our attack is to exclude characters with known exploitable shape patterns. For instance, challenge strings such as "mrnucly" will be less likely to be vulnerable to our attack. To make this method work, it is also necessary to not only examine connection patterns of adjacent characters, but also get rid of those exploitable ones. An additional issue to consider is usability, and therefore a careful control of the distortion level caused by CCT is required.

The above method appears to be applicable to all CCT-based schemes that use random text strings. However, it is inapplicable to schemes such as reCAPTCHA that use English words.

Some other possible defenses include the following. Increasing the number of false loops and breaking some of the valid loops could confuse a loop detection method. Adding random dots in the upper side of challenge images could confuse the dot detection algorithm. The use of font that does not render a cross shape in the characters "t" and "f" could complicate a cross detection algorithm; a similar effect could be achieved by adding random crosses.

6. CAPTCHA ROBUSTNESS EVALUATION: A METHODOLOGY

In the recent years, we have examined numerous CAPTCHAs (including both high-profile ones and less-known ones) and found effective attacks on virtually all of these schemes. In this process, we have been establishing a novel security engineering approach to CAPTCHA robustness evaluation. Our approach applies adversarial thinking skills searching for and exploiting vulnerabilities hidden in CAPTCHAs. In essence, all of our attacks have exploited invariance hidden in the CAPTCHAs, which the distortion and transformation process of each of the Captcha generators failed to eradicate. Exploiting invariants is a classic strategy often used in cryptanalysis. For example, differential cryptanalysis works by observing that a subset of plaintext pairs has an invariant relationship preserved through numerous cipher rounds. Our work demonstrates that exploiting invariants is also effective for examining Captcha robustness.

A brief discussion of selected invariants we have identified in CAPTCHAs was reported in our previous work [7].Our work on the Google scheme and ReCAPTCHA, as reported in this paper, shows for the first time that exploiting shape invariants inherent in characters (such as loops, dots and crosses) and connecting patterns of adjacent characters leads to a novel attack to an entire family of CAPTCHAs that is based on CCT. This work significantly extends our previous understanding of exploitable invariants by identifying some novel invariants and demonstrating how to exploit them.

Here we attempt to provide the first detailed exposition of our "search for invariance" methodology, which has proven effective in practice to reveal critical design flaws and to improve captcha robustness. In particular, we propose a systematic framework that classifies the exploitable invariants that we have identified so far into major categories. Defined in Table 1, our framework provides key insights on security vulnerabilities that text CAPTCHAs should account for.

At the top level, our framework has two categories: *pixel-level invariants* and *string-level invariants*. Pixel-level invariants are all about structural features of challenge images, and they are exploitable by image processing techniques at the pixel level. On the other hand, string-level invariants are about syntax and semantic features of text strings in challenge images, and are independent of any pixel-level features. Typically string-level invariants are created by the linguistic model that a CAPTCHA employs to generate text strings.

Table 1: A framework of exploitable invariants

Category	Invariant Type	
Pixel-level (structural features of images)	Pixel count	
	Color pattern (among pixels)	
	Shape	Shape of character component (or part)
		Shape and position of a whole character (as defined by its bounding box)
		Connection patterns between adjacent characters
		Shape of connected characters (as defined by a bounding box surrounding them)
		Overall geometric features of a challenge string
String-level (syntax and semantic features)	Characters set	
	String length	
	Random string or dictionary word?	

Pixel-level invariants. We have identified three types of pixel-level invariants. The first type is *pixel count*, which is the number of foreground pixels of a connected component (such as a character, a character's part, or a distortion element) in a Captcha image. In our earlier work [4], we identified that in some Captchas, character pixel counts were usually distinct among different characters but remained constant for the same character under different distortions. A simple pixel count attack turned out to effectively break many CAPTCHA schemes at the time. In our attack on the Googe scheme reported in this paper, we exploited pixel counts to differentiate between shared and non-shared components of characters, in order to segment some connected characters. This pixel-count method also helped us to break Microsoft and Yahoo Captchas by differentiating between valid characters and random distortion noise.

The second type of pixel-level invariants is *color pattern* of pixels. Many designers used fancy color schemes for foreground and background pixels in their Captchas to improve usability, to defend against automated attacks, or to do both. The regularity of color patterns in Captcha images helped us to successfully break many designs. A range of case studies is given in our previous work [16].

The third type of pixel-level invariance is *shape invariants*, which are about geometric features of Captcha text strings as rendered in challenge images. Shape invariants can be classified into the following five categories, according to different granularity ranging from a character's part to a text string as a whole.

The first category is geometric features of *character components* (i.e., parts) such as "loops", "dots", and "crosses". Our attack on the Google scheme and ReCAPTCHA heavily exploited such invariants.

The second type of shape invariants is identified from the *overall shape of a whole character* (as defined by its bounding box), sometimes along with the character's relative position in a challenge image. For example, in our attack on Microsoft Captcha [3], we exploited the shape and position of random arcs (i.e. fake characters) to differentiate them from valid characters. The shape of an arc was either too wide or too long to resemble the shape of a valid character. Or an arc often had a relative position different than valid characters did – typically, an arc was closer to the image's boundary. Figure 20 (a) illustrates such scenarios with real examples taken from our previous attack on a Microsoft CAPTCHA: the arcs in the 4th and 6th chunks were discriminated through their relative position with respect to the image boundaries and other characters, and the arc in the last chunk was identified through its overall shape (as defined by its bounding box, which is not drawn in this figure though).

In our attack on the Google scheme, the knowledge of relative position of the shape invariants with respect to their corresponding character category aided the attack's segmentation steps. For example, we knew that the body of the character "i" must be underneath its dot part.

Our attack on ReCAPTCHA exploited such invariants, too. For example, a bounding box with a large width together with a large width-to-height ratio indicated a character such as "m" and "w".

The third category of shape invariants is *connection patterns between adjacent characters*. Our attack on the Google scheme and ReCAPTHCHA exploited this type of invariants, e.g. the "double v" connection pattern for character pairs "oo", "bc", "bd", and "cy" (see Figure 13) in the Google scheme, and the loop shape connecting adjacent characters in ReCAPTCHA. Also, loops with deformed none-circular shapes aided our attack by indicating a connection between characters, for example, in the case of the connection between "ss" (see Figure 6 (b)). On the other hand, white space (gaps) between unconnected adjacent characters can be considered as a special case in this invariant category.

The fourth category of shape invariants is features identified in the *shape of connected characters* as defined by a bounding box surrounding them. Our attack on ReCAPTCHA exploited such invariants. For example, a bounding box with a large width but a small width-to- height ratio indicated connected characters such as "th". We used this pattern to properly segment the characters. Figure 20 (b) shows such an example.

The fifth category of shape invariants is structural features identified in the *overall shape of a CAPTCHA challenge string*. For example, a Yahoo CAPTCHA generated only two types of challenges, either regular or angular (with almost a fixed angle), as shown in Figure 20 (c). We successfully broke this scheme by exploiting such shape invariants [3]. The difference in overall shape between known and the unknown words in the latest ReCAPTCHA (as of May 2011, see Figure 18) allows attackers to customize their attacks only on known words, without bothering with unknown words.

String-level invariants. We have identified three types of string-level invariants. The first is *character set*, which is the set of characters used to compose challenge strings. A Captcha that uses a small character set is more vulnerable to automated attacks than a counterpart using a large character set.

The second type of string-level invariants is *text length*, the length of text strings used in a Captcha. A notable example of this type of invariant occurred in a Microsoft Captcha, which used a fixed length of 8 characters for its text strings. This invariant aided our successful attack [3]. On the other hand, if the text length in a Captcha is fixed and short, and its character set is also small, then this scheme is very likely vulnerable to random guessing attacks. In general, the knowledge of a string length increases the success chance of a segmentation attack based on character estimations.

(a) (b)

(c)

Figure 20. Some examples of shape invariants. (a) the second type, (b) the fourth type, and (c) the fifth type

The third type of string-level invariants is that text strings are always dictionary words, which allows a dictionary attack. A dictionary aided our attack on reCAPTCHA in Section 4, and aided our early attacks on other Captchas [4].

All the invariant types identified in our framework are independent of each other, yet they complement each other. Often, an effective attack has to exploit multiple types of invariants in combination, just as the current paper and our earlier work such as [3, 4] have demonstrated. It's interesting to note that in our defeat of a Yahoo CAPTCHA in [3], exploiting a

correlation between the width of the bounding box of a CAPTCHA challenge string (i.e. the fifth type of shape invariants) and the number of characters in the string (i.e. the second type of string-level invariant) was key to our attack's success.

It is an open problem whether the list of invariants and categories in our framework is complete. We encourage people to identify new types of exploitable invariants, and to extend and refine our framework.

7. GENERAL DESIGN PRINCIPLES

The success of our "search for invariance" methodology in identifying critical vulnerabilities in many deployed CAPTHCAs evidently suggests a new general principle for CAPTCHA design:

A good CAPTCHA should avoid exploitable invariants, which provide shortcuts for effective attacks. The effective methods of identifying exploitable invariants include 1) a manual inspection of randomly chosen challenge images, walking through Table 1 to identify suspicious invariants, 2) automated identification of suspicious invariants by running software tools such as the attacks we have developed, and 3) perhaps more importantly, following the rationale with which we structure our framework might be able to guide the identification of new exploitable invariants. An effective method for removing exploitable invariants in a Captcha is to employ proper randomization techniques in its generator so that the process of distortion and transformation can be refined.

Our extensive study of CAPTCHA robustness also suggests that two general principles established before, as summarized as follows for completeness, still hold.

A good CAPTCHA should be segmentation-resistant. This principle has been established by [5] since 2005 and still holds. However, it is unclear whether it is possible to find a perfect design that is indeed segmentation-resistant and at the same time achieves a good balance between security and usability.

A good CAPTCHA should disable machine-learning attacks or at least make it hard to perform such attacks. This principle was first proposed by one of us -- the second author - and collaborators [12]. Although the principle was initially proposed in the context of image-recognition CAPTCHAs (which require a user to solve an image recognition task), it is also applicable to text CAPTCHAs. For the sake of completeness, we reiterate here some insights that we discussed in [12].

"An intrinsic feature for all machine learning attacks is that they typically rely on empirical data to learn effective discriminative features and decision criteria before becoming effective. The most fundamental solution to deal with these attacks is, therefore, to disable machine learning by making the past challenges uncorrelated with the current or future challenges. In this way, the discriminative features or decision criteria learned from the past challenges would be ineffective to solve the current or a future challenge. This can be achieved by randomly selecting a type and an object of the type to generate a challenge, with both the number of types and the number of individual objects of each type being sufficiently large, infinite ideally, so that it is intractable for the current computing capability.

It is interesting future work to explore whether we can create an unlimited number of segmentation-resistant mechanisms for text CAPTCHAs; or an unlimited number of combinations of segmentation-resistant mechanisms and text-distortion methods. In the meanwhile, an immediate improvement to the robustness of current CAPTCHAs is the following. We can significantly increase the number of segmentation-resistant mechanisms and variants, text distortion methods and fonts that a CAPTCHA generator supports. When a challenge is created, the generator should randomly select one or more fonts, a segmentation-resistance mechanism and a text distortion method. The idea is simple: making it harder for an attacker to do machine learning, we will have a CAPTCHA that is more robust than the state-of-the-art.

8. REFERENCES

[1] von Ahn, L., Blum, M., and Langford, J. 2004. Telling humans and computers apart automatically. Commun. ACM 47, 2 (Feb. 2004), 56-60. http://doi.acm.org/10.1145/966389.

[2] K Chellapilla, K Larson, P Simard and M Czerwinski, "Designing human friendly human interaction proofs", ACM CHI'05, 2005.

[3] J Yan and A S El Ahmad. "A Low-cost Attack on a Microsoft CAPTCHA", 15th ACM Conference on Computer and Communications Security (CCS'08). Virginia, USA, Oct 27-31, 2008. ACM Press. pp. 543-554.

[4] J Yan and A S El Ahmad. "Breaking Visual CAPTCHAs with Naïve Pattern Recognition Algorithms", in Proc. Of the 23rd Annual Computer Security Applications Conference (ACSAC'07). FL, USA, Dec 2007. IEEE computer society. pp 279-291.

[5] K Chellapilla, K Larson, P Simard and M Czerwinski, "Building Segmentation Based Human-friendly Human Interaction Proofs", 2nd Int'l Workshop on Human Interaction Proofs, Springer-Verlag, LNCS 3517, 2005.

[6] K Chellapilla, K Larson, P Simard and M Czerwinski, "Computers beat humans at single character recognition in reading-based Human Interaction Proofs", 2nd Conference on Email and Anti-Spam (CEAS), 2005.

[7] Jeff Yan, Ahmad Salah El Ahmad, "Captcha Robustness: A Security Engineering Perspective," Computer, pp. 54-60, February, 2011.

[8] Chad Houck, Decoding ReCAPTCHA , http://n3on.org/projects/reCAPTCHA/

[9] http://www.theregister.co.uk/2009/12/14/google_recaptcha_busted/

[10] Ahmad Salah El Ahmad, Jeff Yan, and Lindsay Marshall. "The robustness of a new CAPTCHA". In Proceedings of the Third European Workshop on System Security (EUROSEC '10). ACM, New York, NY, USA, pp36-41.

[11] Jeff Yan, Ahmad Salah El Ahmad. "Usability of CAPTCHAs or usability issues in CAPTCHA design. In Proceedings of the 4th symposium on Usable privacy and security (SOUPS '08). ACM, New York, NY, USA, pp44-52.

[12] B. B. Zhu, J. Yan, Q. Li, C. Yang, J. Liu, N. Xu, M. Yi, K. Cai, "Attacks and Design of Image Recognition CAPTCHAs," ACM CCS 2010, pp. 187-200.

[13] Jonathan Wilkins, "Strong CAPTCHA Guidelines" , http://www.bitland.net/captcha.pdf

[14] “Spammers crack Gmail Captcha” ”http://www.theregister.co.uk/2008/02/25/gmail_captcha_crack/

[15] T. Y. Zhang and C. Y. Suen. A fast parallel algorithm for thinning digital patterns. Commun. ACM 27, 3 (March 1984), pp236-239.

[16] Ahmad Salah El Ahmad and Jeff Yan. Colour, usability and security : a case study. Technical Report Series, Newcastle University, England. 2010.

[17] Philippe Golle. “Machine learning attacks against the Asirra CAPTCHA”. In Proceedings of the 15th ACM conference on Computer and communications security (CCS '08). ACM, New York, NY, USA, pp535-542.

[18] Kurt Alfred Kluever. Evaluating the Usability and Security of a Video CAPTCHA. Master’s thesis, Rochester Institute of Technology, Rochester, NY, August 2008.

APPENDIX

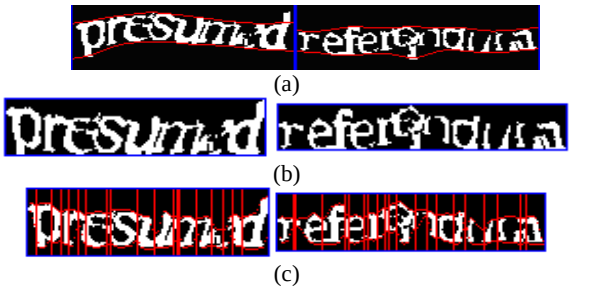
9. The Order of ReCAPTCHA Attack

The order of ReCAPTCHA attack was chosen intuitively as follows. First are dot characters, followed by characters detectable by the “s vertical histogram”, and finally loop characters. The dot character was chosen first because we are confident of its high success rate. Then, we decided to use the “s vertical histogram” algorithm, since many of the characters are detectable using this method, including those with loops such as “e” and “9”, in addition, this algorithm detects characters with broken loops. Last in the order is the detection of loop character, in particular, characters that has a loop but not detectable by the “s vertical histogram” algorithm, such as the characters “o”, “0”, “D”, “A”, and the detection of the connection pattern loop between connected characters.

However, because in ReCAPTCHA attack we are interested in the left and the right sides of characters shapes, rather than the shapes connected components, swapping the order of the loop and the “s vertical histogram” method would not change the success of the attack. This is because the “s vertical histogram” functions similar to the loop detection method and both methods effectively detect the loop left and right side only. As highlighted in Figure 15 (b).

10. The DEFCON Attack

A brief description of the DEFCON attack [8] is as follows. First, we have the image preprocessing including standard binarization, and “wave distortion” removal. The “waving” of the words is removed using the so-called “blanket algorithm”. The blanket algorithm uses a word’s upper and lower contour to estimate the waving; this is done by comparing the slope of neighboring pixels along the contours, and plotting a series of tangents along the top/bottom side of the word. The tangents are then used to estimate the severity of each column along the x-axis. Then, the position of each column in the word is interpolated up or down to straighten the words, additional parameters were used to insure smooth straightening. In Figure 21 (a) (taken from [8]), the red lines above and below the words, represent the series of tangent points, and thus an estimation of each word waving. In Figure 21 (b), show the output after straightening the words. Second, the characters used in ReCAPTCHA are compiled into templates: each template contains the average feature of a character, such as its average width, its average height and the location of its pixels. Third, the image is segmented into multiple “dips” (i.e., chunks in comparison to our attack) using a variation of the “blanket algorithm” [8], again, the word’s upper and lower contours are plotted using the blanket algorithm, then the algorithm searches for valleys in the upper contour, and for peaks in the lower contour, and a vertical line is drawn at each valley or peak location. Figure 21 (c) shows an illustration of the segmentation output, every vertical line represents a “dip”. Fourth, the content between adjacent dips is compared with the character templates. The template matching algorithm could return more than one character with high resemblance for combination of adjacent dips, but initially the character with the highest resemblance is assumed. Finally, a dictionary comparison algorithm is used to verify that the sequence of characters returned by the character templates matching algorithm is an actual dictionary word (other combinations of characters with lower resemblance probability could be used to obtain a dictionary based word).



(a) (b) (c)

Figure 21. Defcon attack. An illustration of the “dips”, each vertical line represents a dip.