Reverse Engineering CAPTCHAs

Abram Hindle, Michael W. Godfrey, Richard C. Holt
Software Architecture Group (SWAG)
University of Waterloo
Waterloo, Ontario, CANADA
{ahindle,migod,holt}@cs.uwaterloo.ca

Abstract

CAPTCHAs are automated Turing tests used to determine if the end-user is human and not an automated program. Users are asked to read and answer Visual CAPTCHAs, which often appear as bitmaps of text characters, in order to gain access to a low-cost resource such as webmail or a blog. CAPTCHAs are generated by software and the structure of a CAPTCHA gives hints to its implementation. Thus due to these properties of image processing and image composition, the process that creates CAPTCHAs can often be reverse engineered. Once the implementation strategy of a family of CAPTCHAs has been reverse engineered the CAPTCHA instances may be solved automatically by leveraging weaknesses in the creation process or by comparing a CAPTCHA's output against itself. In this paper, we present a case study where we reverse engineer and solve real-world CAPTCHAs using simple image processing techniques such as bitmap comparison, thresholding, fill-flood segmentation, dilation, and erosion. We present black-box and white-box methodologies for reverse engineering and solving CAPTCHAs. As well we provide an open source toolkit for solving CAPTCHAs that we have used with a success rates of 99, 95, 61, 30%, and 27% on hundreds of CAPTCHAs from five real-world examples.

1. Introduction

CAPTCHAs are often the lone sentry that guards potentially valuable or abusable resources. CAPTCHA is an acronym for "Completely Automated Public Turing test to tell Computers and Humans Apart"[10]. Many websites and services utilize CAPTCHAs to act as a Turing test for their clients, to ensure that each request comes from an individual human user and is not an attempt by an automated program to acquires resources in bulk (Figure 1). Essentially, a CAPTCHA is a strange kind of lock, one that is designed to be easily broken by human visual pattern matching, but not by means of automated software. That is, CAPTCHA design is an exercise in creating a breakable lock.

Unfortunately CAPTCHAs limit not only automated spam bots from using resources, their use is an impediment to anyone who is visually impaired, or anyone who has to rely on audio screen readers, braille screen readers or alternative browsers. Even if an audio CAPTCHA is provided there are still users who are discriminated against. Thus there are legitimate reasons to automate the solving of these CAPTCHAs, whether the purpose is humanistic or malicious.

Solving CAPTCHA is an Hard AI image processing problem in the general case [10], thus it is hard to create an all encompassing CAPTCHA solver, but it is easy to create a solver for a family of CAPTCHAs. There are an infinite number of methods to produce CAPTCHAs but CAPTCHA generation is also an Hard AI problem (in the general case) [10]; this makes CAPTCHAs similar to the SPAM problem: a continuous interaction between adversarial elements trying to tune their techniques to defeat each other's defences.

The CAPTCHAs we consider in this paper are primarily bitmap depictions of text that the user is meant to recognize and reproduce. Audio CAPTCHAs and picture CAPTCHAs are not dealt with in this study. For sake of brevity, within this paper we will use the term *"captcha"* to refer only to visual textual bitmapped CAPTCHAs.

Captchas [1] are generated by software, thus they are relatively deterministic. Their output might vary wildly, but the output, the instance of the captcha, was created by following a deterministic process. As we mentioned above, general captcha generation is a Hard AI problem, captcha generators avoid this complexity by generating specific families of captchas by following a deterministic process. This process and this software can be reverse engineered in both black-box and white-box (source available) settings. Thus target captchas can be solved by reverse engineering and re-

[1] From this point on, we will use the more common lower-cased spelling of the term.

implementing the captcha generator. A target captcha may also be solved by exploiting the weaknesses of the operations identified while reverse engineering the captcha.

Our contributions include:

- A methodology for reverse engineering the process of captcha creation
- A general methodology for solving captchas
- A method of solving captchas by leveraging an existing implementation
- A set of tools for captcha solving

1.1. Previous Work

Von Ahn et al. [10] formalized the idea of CAPTCHAs as an automated Turing test that leveraged Hard AI problems. We have previous work that briefly describes some of our captcha solving efforts [5]. Much of our work was motivated by the work on the application of shape matching to the EZ-Gimpy captcha [7]. The shape matching algorithm was used for character recognition. It relied on matching the points of the contour of a shape to points of shapes in the database and then finding the best permutation of matching points, measuring the distance and then choosing the class of the closest matching candidate.

Other work focused solely on segmentation [12], which dealt with noise removal via supervised learning and rules. Others further applied machine learning techniques to aid segmentation and character recognition [3]. We have adopted the use of K-Means clustering [6] to segment captchas as described by our colleagues Caine et al. [2].

2. Common Properties of Captchas

The constraints that are important to every captcha are:

Readable: the captcha must be easily read and decoded by humans.

Unguessable: The captcha message cannot be guessed at random with any real confidence.

Order-able: Characters are read left to right, top to bottom (exceptions could include Hebrew or Arabic captchas). If a captcha is readable, its character ordering should be apparent.

These constraints are important because difficult captchas can dissuade potential customers, which is not the intent of using captchas. Given these constraints some features and properties of common captcha implementations include:

Bitmap fonts in fixed positions: Many Captchas consist of text in a fixed position written in a bitmap font. Although they are easy to break, they are easy to generate and thus very common while still serving as a minor deterrent to spammers.

Static background: Captchas that use static backgrounds are easy to spot because the background is always the same color or texture.

Random placement: Many captchas use randomly placed characters instead of characters in static positions as it can complicate segmentation and recognition.

Background/foreground noise: Captchas often deploy background noise and overlapping foreground noise. Noise often consists of random pixels, lines and other objects. This makes comparison less deterministic.

Linear transformations: Captchas often warp their characters via linear transformations such as rotation or skew. These characters are more difficult to match because they need to be normalized first.

Non linear transformations: More advanced captchas rely on non-linear transformations and warping because they are harder to normalize. Non-linear transforms include projecting characters onto surfaces, smearing and warping.

Dripping/Fuzzy text: Some captchas employ creative distortions by obscuring individual characters with smears and spikes that often renders text difficult to read and to match.

Layering: Most importantly many of these captchas follow some general pattern of composition: the layering of these above techniques, the use of overlays, and transformations. Identifying how a captcha is layered is similar to discovering how the captcha was created.

3. Methodology

The main methodology for solving a captcha is first to analyze the captcha and determine the steps of its generation process, find relevant counter-techniques, then to configure a specific solver from this information. The specific captcha solver will exploit the weaknesses of the methods used to generate that family of captchas.

3.1. Reverse Engineering a Captcha

The knowledge of common captcha implementation strategies allows us to reverse engineer the process of captcha creation. This helps us choose and configure the appropriate tools to solve the captcha. In some cases, if a captcha can be reproduced, then instances can also be solved by comparing them to instances of the captcha generated during a parameter search (as explained in Section 3.3). A general methodology for reverse engineering a captcha is to identify the following operations and order them by their layering:

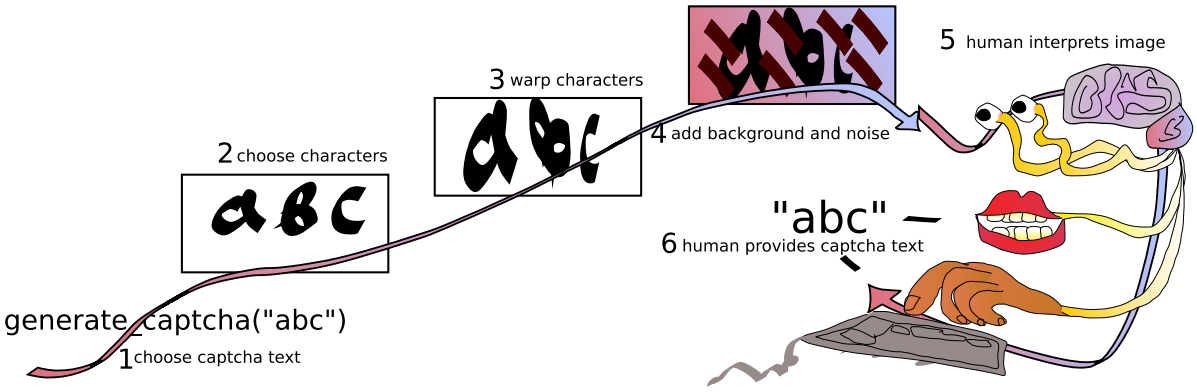


Figure 1. CAPTCHA Creation and Human Deciphering

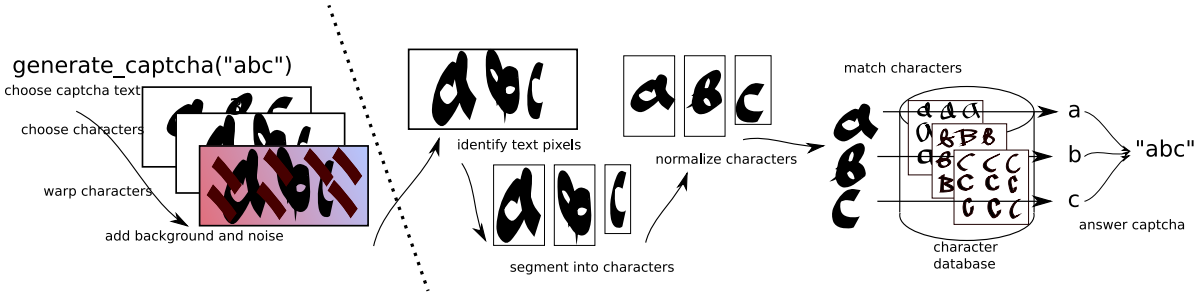


Figure 2. Model of Computer CAPTCHA Solving

- Layering: discover operations and overlays
- Background: discovery and filtering
- Noise: method and possible removal
- Text: method and properties of the text
- Transforms: method, what objects or layers are transformed, and possible reverse transforms

Layering refers to the general pattern of captcha construction. Layers consist of bitmaps, text, vector graphics, transforms and noise. Layers are usually placed on top of each other cumulatively (assuming some transparency or opacity) or in the case of transforms, as operations in a sequence. In many cases the sequence of operations applied is analogous to layering. This is similar to how layering is used in photo-editing software like Adobe Photoshop or the GIMP. One difficulty is in determining when the text is added, although if one assumes that layers are added cumulatively, it is often not hard to see if the text is overlapped or transformed by anything such as line noise or a rotation. Layering is often given away by overlapping objects or a difference between background and foreground elements.

Backgrounds are added mainly to cause confusion for solvers as they usually add noise that makes analysis more difficult. Often backgrounds are a single color or a static image, these can be identified and ignored by looking for common pixels among a set of captchas. Background noise could include lines and shapes meant to cause edge noise in an effort to confuse edge detection, or random pixels, that are the same color as text, spread about to confuse segmentation. It could also include false letters drawn differently as to hinder segmentation or character recognition. The background could also mimic the colors and textures of the characters. Backgrounds are obvious because they are the negative space around characters.

Noise is commonly added after text characters and background layers are applied, sometimes noise is only added to the background. Often noise consists of randomly placed grey pixels, random lines, and shapes that overlap the characters and background. Noise is often visually identifiable by its presence. Image compression can also add noise. Noise is often removable by thresholding, erosion, and dilation.

Text characters are drawn in captchas in a variety of ways. Captchas consisting of untransformed text generated from bitmapped fonts, placed randomly or statically, are easy to solve as segmentation is often not required because the cost of bitmap comparison of N bitmaps per each pixel is so low it renders segmentation unnecessary. Gen-

erally one can order characters by their x coordinates since captchas must be readable and orderable. To determine the layering of the text, check if the text is occluded or transformed independently of a background, then it is obviously in a sublayer. Often text pixels are easy to identify because they share a similar set of properties, distinct from the background.

Transformations are applied either per character, over all the text at once, or over the whole image. Transformations come in two main categories: linear and non-linear transformations. Linear transformation include skew, rotation, and projection. Non-linear transformations often include projecting the text onto a bumpy surface like a flag or warping. Linear transforms can be normalized by Principle Component Analysis (PCA), while non-linear transforms require the more complicated Independent Component Analysis (ICA). If these transformations are independent of the background or noise then they were applied per character or just to the text layer. Global transforms are identifiable if the background suffers from the same warping as the text characters. If characters are transformed independently it is a good sign that the transformation was applied per character.

Figuring out the layering and operations is often the most important step because it indicates an order of operations which is the process of captcha creation itself. It can also help identify those operations that are weak against counter measures like PCA or fill flooding.

3.2. Solving a Captcha

Solving a captcha usually has four main steps: image clean up, text pixel identification, segmentation, and character matching. Figure 1 illustrates how a human solves a captcha, while Figure 2 illustrates the process that a computer program goes through to a solve a captcha. The steps of this process and the important algorithms related to each step are:

1. **Image Clean Up** removes noise that could harm later steps.

   Erosion / dilation can sometimes clean up background noise and reconnect disconnected characters; the effectiveness depends on the color scheme used.

   Thresholding can clean up, and often remove a background before text pixel identification.

   Lone pixel removal reduces noise by removing potential text pixels that are relatively isolated from other potential text pixels.

2. **Text pixel identification** categorizes individual pixels as either text or non-text.

   Edge detection , which is a computer vision technique to detect hard edges in an image, is used to detect the hard edges of characters if the background is soft.

   Thresholding looks for values above or below a certain color or luminosity. It often works because many captchas have text that is an extreme color (black or white).

   Fill Flooding works by flooding a color or value recursively in multiple directions so that one can look for continuous areas large enough and uniform enough to be characters.

3. **Segmentation** , using a variety of means, parses the text pixels into separate character segments.

   Fill Flood Segmenting utilizes fill flooding to separate and segment text pixel regions.

   Weight Segmenter segments along the x axis by the counts of text pixels per column.

   Box Segmenter grows segments around text pixels until no more text pixels are adjacent.

   K-Means Segmenter segments text pixels via K-Means clustering [6].

   Shrink and Fill segments by applying erosion or shrinking and fill flooding repeatedly until the optimal number of segments have been found.

4. **Character Matching** matches the segments against characters in a database, often using a normalization step (see Figures 7(a) and 7(b) for an example of character databases). The following techniques are employed in character matching:

   Centering normalizes characters; often they are centered in a bitmap and scaled to a certain size so that they can be compared against other candidates in the database.

   Principal/Independent Component Analysis is often used to normalize the contour, shape data, or pixels of a characters. PCA is used to normalize against linear transformations, while ICA is used on non-linear transformation.

   Nearest Neighbor is a simple machine learning technique to compare vectors and choose the class of an instance from the classes of the nearest K instances from the database.

   Shape Matching , described in [7], matches the contour of a character against those in a database via the best matching permutation of paired points.

   Skeletonization reduces the dimensionality of the data and eases comparison by generating skeletons of characters.

3.3. Solving a Captcha by Cloning

One possible method of solving a captcha, where the source code is available (open source web applications) or the captcha is easy to re-implement (e.g., you have already reverse engineered the captcha), is to parametrize the generators to produce captchas that one can compare against a target instance captcha. This would allow for a nearly linear time search of valid characters (per character, try each character in the alphabet, find the best match and move on).

A method for solve-by-cloning is:

1. Reverse engineer the captcha generating code: either re-implement it by reverse engineering or find the source code.
2. Parametrize the captcha so the previously random values for properties such as distortion, text placement, and background are now parameters and potentially search-able.
3. Preprocess the captchas to remove noise.
4. Search through generated captchas by character positions from left to right, generating all captchas with the same prefix but a different one letter suffix, keep those which best match, extend the prefix with the good match and continue on to the next character. Eventually the prefix will converge on the solution to the instance of the captcha. Repeating this step might improve accuracy.
5. Find the best fit of generated captchas by parameter search

Thus by parameterizing and searching/estimating the parameter space of the target captcha, an instance of a captcha can be solved by similarity measures. This method is efficient in the sense that only $N * k$ comparisons are needed where N is the number of positions and k is the size of the alphabet. One problem encountered when solve-by-cloning is that often the placement or size of different characters might cause the text to be centered differently. Sometimes the entire text is centered in the image so a search over all possible text might require some sort of centering correction otherwise matches will be error prone. See our solve-by-cloning solution to the Watercap captcha in Section 5.2.5.

3.4. Developing a Specific Captcha Solver

The general process for developing a captcha solver is as follows:

1. Get a sampling of target captchas.
2. Solve some of the sampled target captchas manually.
3. Label the solved target captchas.
4. Develop the captcha cleaner (see section 3.2).
5. Develop the captcha segmenter (see section 3.2).
6. Segment some of the solved captchas.
7. Create a database of labeled segments (train on examples).
8. Develop the captcha solver (see section 3.2).
9. Test trained and untrained solved captchas against captcha solver
   (a) If the accuracy is acceptable, then continue.
   (b) Else re-integrate newly solved segments and resolve;
   (c) If accuracy is still too low, improve the solver and segmenter.
   (d) Else if accuracy is still too low, improve the cleanup algorithm and re-segment from the beginning.
10. Test against unsolved captchas, improve the character segment database as needed.

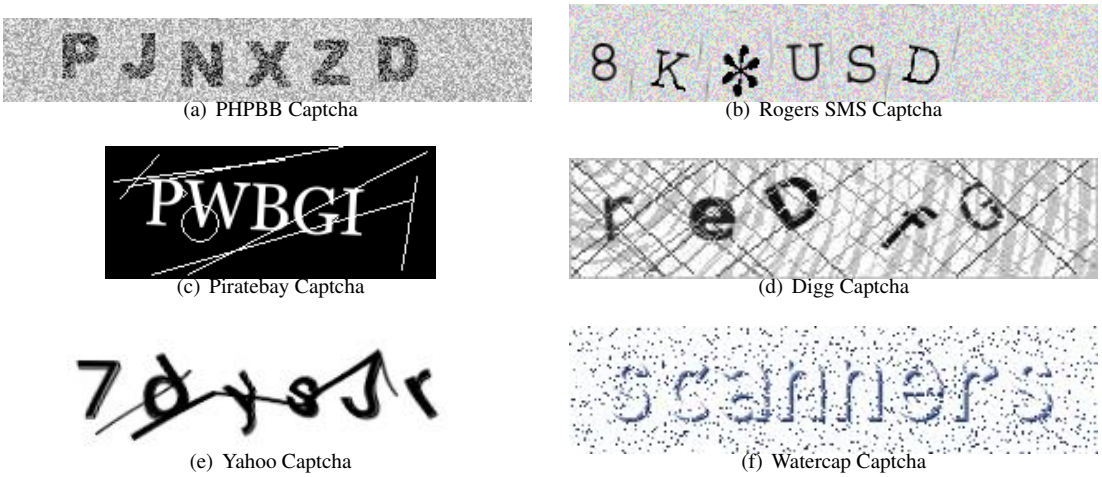
One of the most fruitful and important steps in captcha solving is the integration of solved and miscategorized segments into the database. Often more and better data will result in a more accurate solver.

4. Captcha Solving Algorithms

Many of these algorithms are quite useful for preprocessing captchas, segmenting them, normalizing them and matching segments. Many of these image processing algorithms are parallel algorithms. They operate on one image and produce another new image. Often if an image processing algorithm does not read from one buffer and write to another it causes cataclysmic problems like disconnected characters and loss of information. Many of these image processing algorithms rely on knowing about the pixels around a target pixel (usually a 3x3 square around the pixel in question, sometimes more) and thus these algorithms need the original image to be untouched while they write to a new image.

4.1. Erosion and Dilation

Erosion and dilation are essentially the same transform but dilation makes bright colored areas bigger while dilation makes the dark colored areas bigger. These image processing operations are commonly used thicken or thin text characters. For example if dark text characters are placed together snugly on a bright background, dilation, by growing the brighter background, might separate the characters



(a) PHPBB Captcha (b) Rogers SMS Captcha

(c) Piratebay Captcha (d) Digg Captcha

(e) Yahoo Captcha (f) Watercap Captcha

Figure 3. Examples of Captchas

better so that we can apply a fill segmenter, whereas thin dark anti-aliased characters that are almost merging into the background can be thickened by using erosion. Erosion and dilation can also be used in a positive sense where objects (text pixels) are posed as positive space, then erosion and dilation make objects shrink or grow.

4.2. Thresholding

Thresholding is the simple act of partitioning pixels into two groups, text and background, by a value such as color or magnitude/brightness. Pixels matching values above or below a threshold are labeled text or background. Thresholding is used to help segment characters and find character pixels.

For example, if the text characters of a captcha are pure white, then with a threshold of 99% brightness, anything darker would indicate a background pixel. Thresholding works well on many captchas, such as the Digg captcha, because the background colors are sufficiently darker or lighter than the text characters.

There are disadvantages to thresholding: for example, it can disconnect parts of a character making fill-based segmentation difficult, especially if the text is anti-aliased.

4.3. Shrinking and Pixel Removal

Shrinking is often seen as erosion but it can also be performed by removing potential text pixels that do not have enough neighbors. Shrinking is quite useful to remove line noise because often line noise is applied later in the generation process and is often not anti-aliased. Shrinking is often done after thresholding, as it can remove islands of lone text pixels. A common algorithm for shrinking is to remove every text pixel that has fewer than k text pixels around it. This is done in parallel and the output is written to a different buffer.

Lone pixel removal often helps with segmentation. The algorithm is quite simple: if a pixel is not surrounded by other text pixels, then that pixel is removed in the output buffer.

Shrinking and fill flooding can be combined to make a segmenter that handles characters that are too close together until only K candidate segments remain.

4.4. Fill Flood Segmenter

Fill flooding works well for identifying text pixels as well as segmentation. The fill flood segmenter looks for regions of continuous color by fill flooding an unique value per each pixel. Once each region is fill flooded the top candidate regions are evaluated and filtered depending on size requirements. In some cases the background will not produce regions that are fill flood-able. In some cases the text pixels have already been found so fill flooding only needs to be applied to the text pixels themselves.

The fill flood follows a simple recursive algorithm of coloring in pixels of a similar uncolored value with the current fill color in either four or eight directions (North, West, South, East, and North-west, South-west, South-east, North-east).

The benefits of fill flooding are that it is quite fast and it can handle text that has overlapping ranges, although it does not work well on text that touches or truly overlaps with the same color. Fill segmenting sometimes benefits from anti-aliasing because it defines segment borders and separates characters. Fill flood segmenters do not work well on characters that are not continuous. Characters composed of dots can defeat this segmenter as well as characters that have non-uniform colors like gradients.

4.5. Weight Segmenter

Once text pixels are detected one can slice the segments by the dips produced in the histogram of the text pixels per column. This is similar to making all the text pixels fall from their positions and stack on top of each other then drawing segmentation lines in the dips between the larger lumps.

This segmenter works well in cases that have a lot of line noise but where the characters do not overlap. This segmenter is essentially half of a common region finder segmenter used in OCR. This also has been called vertical segmentation [12].

4.6. Box Segmenter

The box segmenter attempts to increase a segment size of a possibly disconnected cluster of segments which overlap simple ranges. The box segmenter algorithm is simple: start at a text pixel, put a box around it, then keep extending the edges of the box until the edges do not overlap a text pixel. Keep extending each edge of the box until no edge is adjacent to any text pixel.

The box segmenter works poorly on overlapping characters but works well on disconnected characters. For example, the Digg captcha has a lot of line noise; consequently, the normalization process can separate parts of the characters so that fill flooding does not work but the box segmenting often does.

4.7. K-Means Segmenter

K-Means segmentation [2] is the use of K-means clustering [6] on text pixels to determine which pixels belong to which character. One provides the number of clusters to look for, K, and the K-Means algorithm tries to build the number clusters suggested. The disadvantages of K-Means is it unstable and often requires multiple runs just to be confident in the clusters chosen. Also large characters like M's and G's and small characters like i's and l's sometimes confuse the clustering. K-Means has been successfully used to segment the Ticket Master captcha [2].

4.8. Microsoft Captcha Segmentation

The Microsoft captcha segmentation algorithm, as discussed by Yan et al. [12], combines methods to segment noisy captchas. They have built an arc identifier that identifies arcs which are not characters. Using fill floods they then find candidate regions by using a decision tree. Noise arcs are removed and candidate segments are identified.

4.9. Principal Component Analysis

Principal Component Analysis (PCA) attempts to find a vector with the most variation through a multidimensional data set. Characters often are normalized via PCA. There are two main ways to apply PCA, one is to operate on a high dimensional data set consisting of the pixels of the characters as a vector, the other is to act on a two dimensional data-set of the contour points (the points along the edge of the characters) of a character. By dealing with the contour one can normalize the characters that have been linearly transformed.

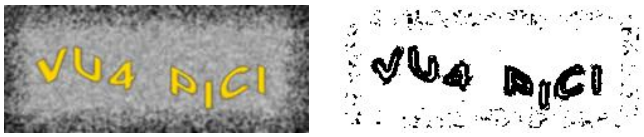


Figure 5. Edge detection example

Figure 4 demonstrates the application PCA to the contours of 360 rotations of A's and F's. Note how the rotations A's and F's map back to the four quadrants of the Cartesian plane.

4.10. Edge Detection

If characters are filled with noise but are differentiable from the background, edge detection [1] can be used to detect the contour of the characters. This is found by analyzing the difference in color or luminosity between a pixel and its surrounding pixels.

4.11. Rosenfeld Skeletonization

Skeletonization attempts to find the connected structure of a continuous block of pixels. It seeks to represent a skeletal structure of an object and maintain certain properties of that object. Common properties include: keep the sharp end points of the object, stay inside of the object, keep intersections, and keep connected objects connected. There are many kinds of skeletonization and each has different properties. Skeletonization can be used as a dimensionality reduction step that seeks to keep various geometric properties. It is also good for generating small geometric graphs of thicker objects.

The skeletonization algorithm we used was the Rosenfeld Skeletonization algorithm [9], it tries to maintain thinness and connectivity while being a skeleton within the object itself. Figure 6 illustrated Rosenfeld skeletonization applied to a captcha.

4.12. Nearest Neighbor

Nearest Neighbor is the method of classifying a query instance based on its distance to K nearest neighbor instances in the database. Distance is calculated via metrics such as Euclidean distance. It is assumed that the majority class of the K neighbors will likely be the class of the query instance. Nearest neighbor works well when one describes a segment as an N dimensional vector. The simplest case of nearest neighbor is 1 nearest neighbor.

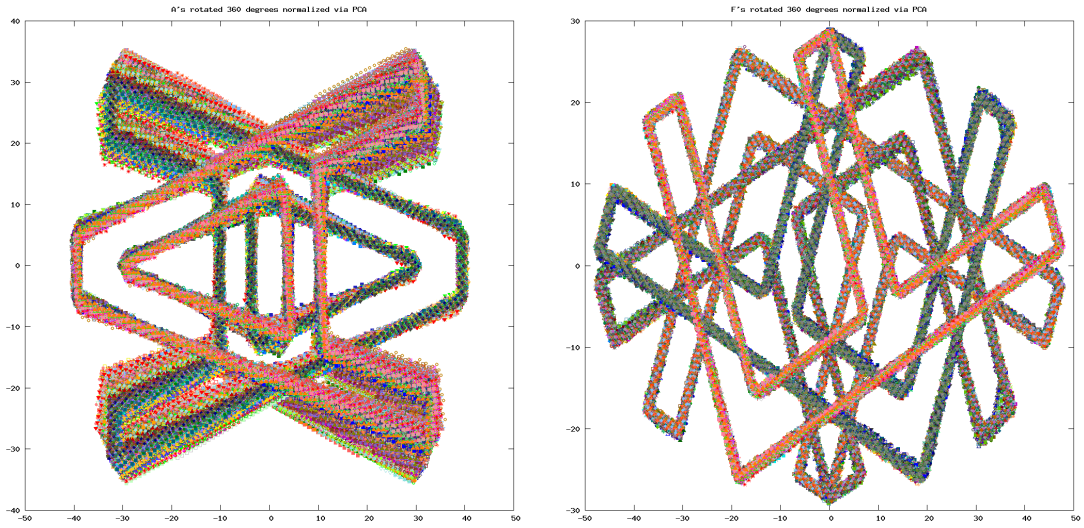


Figure 4. PCA Applied to 360 rotations of the letters A and F

Figure 6. Rosenfeld skeletonization of a captcha

5. Case Study

We studied multiple captchas and solved most of them with reasonable accuracy (30% or better) using our toolkit and our utilities.

5.1. Utilities and Tools

Our tools [4], consist of a library of useful captcha solving functions written mostly in OCaml. The library provides much of the functionality described in the previous sections. General purpose infrastructure includes font loading routines, image routines, transformations and algorithms such as PCA.

We also have utilities that aid in the process of reverse engineering a captcha as well as labelling segments. Many of our utilities are command line versions of the previously mentioned algorithms so we can prototype techniques for cleaning or segmenting a captcha.

To handle segmentation we have two main tools: the segment mover and the segment namer. The segment mover is used when there is a corpus of already solved captchas, where the image files are named by their solution (e.g., `abc.jpg` for "abc"). Segment mover takes the segments produced by the segmenter that has been configured, and then moves the segments to the appropriate letter collection they belong to, assuming the segmenter was correct. For example, the third segment of `abc.jpg` would be moved to the `c` collection. These collections then need to be inspected, and the errors reclassified.

The segment namer takes un-labeled segments, displays them and waits for user input to determine which letter the character represents. This produces a shell script that will move all the appropriate segments into their respective letter collections.

5.2. Captcha breaking results

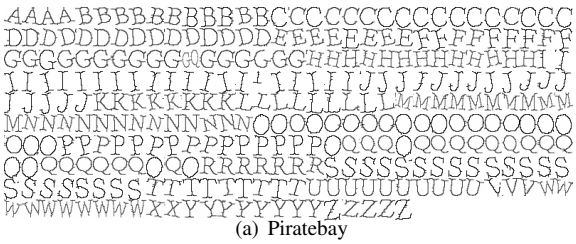
We will now discuss the captcha families that we tried to break and summarize our success with each: PHPBB at 99%, Rogers at 95%, Piratebay at 61%, Digg at 30% and Watercap at 27%/93%.

5.2.1. PHPBB Captcha

The PHPBB Captcha is used by the PHPBB forums software, which is an Open Source web forum. PHPBB's captcha is quite simple (see Figure 3(a)), consisting of a grey background with randomly placed but uniformly spaced black letters, all covered by a layer of grey noise. PHPBB's font is static and its characters are un-warped. We found that simple box segmentation and bitmap comparison can be used to solve the captcha with almost 99% accuracy as the noise left more than 50% of the text pixels in place. We achieved 99% accuracy on over 100 captchas.

5.2.2. Rogers SMS Captcha

The Roger's SMS captcha is particularly interesting (see Figure 3(b)), consisting of rotated characters on a noisy grey



(a) Piratebay

(b) Digg

Figure 7. Samples of character databases for the Piratebay and Digg

background. This captcha was used for guarding a SMS sending application. To defeat this captcha we use thresholding and dilation to segment the captcha. We found that the text pixels were obvious, as they were black and nothing else was black. We then sampled the points of the contour of the characters, processed those 2D points with PCA and then matched against our database using nearest neighbor. We found that shape matching did not help that much although it reportedly worked well on other captchas [7]. We achieved greater than 95% accuracy on over 150 captchas.

5.2.3. PirateBay Captcha

The PirateBay captcha is a rotated font captcha where the characters might overlap in range (see Figure 3(c)). There is some geometrical line and circle noise added as well. To solve the PirateBay captcha we tried to remove the lines and circles by removing white pixels that were not surrounded by many white pixels. Then we skeletonized the remaining white text pixels using Rosenfeld skeletonization (see section 4.11). Taking these skeletons (see Figure 7(a)), we PCA'd their points and then did a nearest neighbor comparison to known samples. This resulted in 61% accuracy on 100 captchas.

5.2.4. Digg Captcha

The Digg captcha is quite noisy (see Figure 3(d)). It has a grey background of concentric circles which make edge detection slightly more difficult. Multi-tone line noise intersects the characters, segmenting them into multiple parts. This makes fill flooding difficult, multiple passes of dilation and erosion can be used to remove most of the line noise. K-means segmentation worked well against the Digg captcha. Then the clusters of pixels were abstracted as points and then run through PCA. The PCA'd characters were compared against the database using nearest neighbors. This results in a solver that had 30% accuracy on 540 captchas.

5.2.5. Watercap Captcha

We naively defeated the Watercap captcha [8] (see Figure 3(f)) by applying solve-by-cloning. We made an image distance program that outputs the image distance between images, then we took the Watercap captcha code, and tooled it for a linear search. Using a 20 line PERL script we had it call the Watercap captcha code to generate candidate captchas, then our image distancer (based on the PHPBB code) determined the closest captchas. Using this method we solved 27% of 100 randomly generated Watercap captchas of dictionary words 8 characters in length; 2 passes of the algorithm corrected most of the previous mistakes, resulting in 93% accuracy. A real solver or more preprocessing would have done better.

5.2.6. Yahoo Captcha

Currently we have yet to confidently solve the Yahoo captcha (see Figure 3(e)). Separating the non-textual lines out of the captcha has proven to be difficult. As well the warping of the characters makes character matching difficult as it is non-linear. Some success has been made by others, who employ neural networks for segmentation and character recognition [11].

6. Discussion

Based on our experience with the case studies we have some recommendations and some comments about the general security of captchas, and some future directions to investigate.

6.1. Suggestions for Captcha Design

If one still wants to use captchas for marginal security (a security effort easily beaten by human effort), one should attempt to make a captcha that is so difficult for programs to solve that it serves as a deterrent. Difficult captchas are hard to segment and hard to match. We have provided some recommendations that improve the difficultly of solving such captchas:

Non-linear transformations: As explained before linear transforms are easily corrected and normalized by techniques such as PCA. Other non-linear techniques are more difficult to implement thus using non-linear transformations can impede captcha solving.

Non fill flood-able letters: Characters that are filled with a texture or a gradient will stop fill flood segmentation and often defeat thresholding. The more dynamic the characters are color-wise, the harder they are to identify.

Use more characters: Use more characters and more kinds of characters (uppercase, lowercase, numbers, punc-

tuation). If one can only match characters 90% of the time, more characters will reduce total accuracy.

Limit the number of captcha attempts: Only allow a limited number of captcha attempts at a certain rate.

Similar to the background: If the text is nearly indistinguishable from the background it will be harder to match.

Non continuous characters: Characters that rely on Gestalt principles like closely spaced dashes and dots will defeat many of the common segmenters.

Overlapping characters: Characters that overlap or are embedded in each other often cause a myriad of segmentation problems.

6.2. Ethical Considerations

Some readers may be concerned about the ethics of captcha solving. Indeed, any research on security-related technologies requires considering the ethical ramifications of the work.

We consider that research on captcha solving is ethical. First, we note that many captchas in current use are poorly engineered and easily solved. The goal of captcha design is to create a lock that is easily broken by a human but hard to break by automated software. If we have shown that the state-of-practice fails to meet these goals, then clearly more work needs to be done. We note that we have also provided a set of recommendations for captcha designers to improve their techniques.

Second, we feel that the risk posed by publicizing this work is small. The resources guarded by captchas are meant to be given out, just not in bulk.

Third, captchas provide an unnecessary barrier to accessibility by limiting automation. This means those who rely on alternative means to use websites or software will be denied access to these resources.

We hope that this paper demonstrates that current captcha generation practices yield poor security and poor locks, which can be relatively easily broken; they also unintentionally limit users of alternative browsing methods.

6.3. Future Work

We would like to evaluate more captcha solving techniques and tools such as: hierarchical machine learners, color subspace analysis, contour analysis, Fourier transforms, etc. As well we wish to study how to reverse engineer audio CAPTCHAs.

7. Conclusions

We have shown that the properties of 2D image generation enable us to identify common operations and patterns used to generate captchas. These operations can be reverse engineered from instances of the captchas themselves. The combination and ordering of these image operations is often identifiable, enabling the underlying creation process to be discovered, thus potentially allowing a family of captchas to be solved or re-implemented.

We have confirmed the results of previous work that captchas are often solvable, and we have also provided a more general framework for reverse engineering a captcha in order to either solve or re-implement it. We have solved a number of captchas ranging in use, purpose, implementation and difficulty.

We have followed and provided a methodology for reverse engineering and solving families of captchas. We also provided a tool-chest of algorithms for captcha solving, each illustrating its appropriateness.

We have provided recommendations for making better captchas, based upon previous work, our own personal experience, and observations: use non-linear transformations. We have also released the code and utilities that we used for captcha solving[4].

Acknowledgements: Thanks to NSERC for a PGS D scholarship, and to Philippe Vachon and Alan Caine for fruitful CAPTCHA discussions.

References

[1] H. Bunke and W. P. *Handbook of Character Recognition and Document Image Analysis*. World Scientific, 1997.

[2] A. Caine and U. Hengartner. The AI Hardness of CAPTCHAs does not imply Robust Network Security. In *Trust Management*, pages 367–382. 2007.

[3] K. Chellapilla and P. Y. Simard. Using machine learning to break visual human interaction proofs (HIPs). In *NIPS*, 2004.

[4] A. Hindle. Free software open source captcha breaker: Adversarial OCR. http://churchturing.org/captcha-dist/, 2006.

[5] A. Hindle. Poster: Captcha Breaking: The Visual Adversary. In *UWaterloo GSRC*, 2007.

[6] J. B. Macqueen. Some methods of classification and analysis of multivariate observations. In *Berkeley Symposium on Mathematical Statistics and Probability*, 1967.

[7] G. Mori and J. Malik. Recognizing objects in adversarial clutter: breaking a visual CAPTCHA. In *Computer Vision and Pattern Recogition*, 2003.

[8] P. Simakov. Cracking Watercap Captcha in 24 Hours. http://www.softwaresecretweapons.com/jspwiki/cracking_-watercap_captcha_in_24_hours, 2007.

[9] R. Stefanelli and A. Rosenfeld. Some parallel thinning algorithms for digital pictures. *J. ACM*, 18(2):255–264, 1971.

[10] L. von Ahn, M. Blum, N. Hopper, and J. Langford. CAPTCHA: Using Hard AI Problems for Security. In *Advances in Cryptology EUROCRYPT 2003*, page 646. 2003.

[11] J. Wane. Yahoo! CAPTCHA is broken. http://network-security-research.blogspot.com/2008/01/yahoo-captcha-is-broken.html, 2008.

[12] J. Yan and A. S. E. Ahmad. A Low-cost Attack on a Microsoft CAPTCHA. Technical report, School of Computing Science, Newcastle University, UK, 2008.