**Decoding reCAPTCHA**

Chad W. Houck

[0xn3on@gmail.com](mailto:0xn3on@gmail.com)

**Abstract**

Due to the prevalence of spammers on the internet CAPTCHAs have become a necessary security measure. Without a CAPTCHA in place a system is incapable of knowing whether a human or an automated computer is executing a request. Currently one of the most widely implemented versions of this system is Google’s reCAPTCHA due to its robustness thus far. This paper illustrates techniques to defeat this system which has been trusted to secure websites such as Twitter, Facebook, Craigslist, and many others, as well as methods to secure it further. The efficacy of the techniques outlined herein is at a very conservative figure of ten percent, which is more than enough for an applicable exploitation of the system.

**1. Introduction**

It has been proven time and time again that as long as human’s are capable of reading a CAPTCHA, technology can be created to defeat it. To thwart this problem CAPTCHAs are becoming more and more complicated with heavy distortions to make the decoding process more difficult. The purpose of this research was to develop a method to defeat the state of the art techniques implemented by Google’s reCAPTCHA system. In doing so developers and implementers of this system can become aware of its flaws and be prepared for any future exploitation by taking proactive measures to ensure their systems are secure. Decoding the CAPTCHA was accomplished by implementing various algorithms to remove the distortion of the image to a reasonable degree. Then segment the image into various portions to narrow down the possible characters. The segments are then analyzed to determine the most closely matched characters and subsequent words. These methods are described in detail throughout the following sections, and validated by experimental results.

**2. Related Works**

According to research published by Jonathan Wilkins he successfully solved reCAPTCHA back in January 2010. Google claims however that he was using older images from 2008. He refutes that statement by showing statistics from more recent CAPTCHA images. Wilkins claims that the most complicated forms of reCAPTCHA are words with lines through them. reCAPTCHA has since drastically changed the distortions placed on the words back in January. According to Google, “We’ve found reCAPTCHA to be far more resilient while also striking a good balance with human usability, and we’ve received very positive feedback from customers.” (allspammedup.com) Researchers at UC Berkeley has discussed various methods for breaking CAPTCHAs. They use the “general framework that we used to compare images of everyday objects and even find and track people in video sequences” to examine and decode CAPTCHAs ([www.cs.sfu.ca](http://www.cs.sfu.ca)). The research is unfortunately limited to simpler forms of CAPTCHAs due to being so generalized.

**3. Paper Outline**

The remainder of the paper is organized as follows. In section 4 we discuss removing the multiple layers of distortion added to the words placed by reCAPTCHA. Section 5 describes the methodologies used for character recognition along with a dictionary attack to find the most closely matched word. In Section 6 the experimental results are detailed explaining the efficacy of the algorithms in detail. Section 7 suggests solutions to deterring automated decoding of CAPTCHAs along with the methods described in this paper. Section 8 concludes the paper , and a bibliography in Section 9.

**4. Removing Distortion**



Figure 1

A sample CAPTCHA taken from a reCAPTCHA server on Mar 05, 2010 is labeled as Figure 1. One of the more noticeable differences of reCAPTCHA from typical CAPTCHA systems is the dual word request. This is necessary for their system to provide not only a Turing Test that secures a system from an automated computer by ensuring that a human is typing the image but to also help digitize books. By issuing two words reCAPTCHA can send one which is the verification word, and one that their optical character recognition (OCR) software cannot decode. They trust that if the user is capable of correctly entering the verification word then they correctly entered the unverified word. One important feature of this system is that the user entering the CAPTCHA text need only enter the verification word while entering any arbitrary text for the other. Both words are presented randomly each time to deter regular and savvy users of the system from attempting to save time by skipping or simply misrepresenting the unverified word. A few other important notes to make about this system is that a user can replace one character of the verification word with anything they like or they can leave it out entirely and reCAPTCHA will still accept it. The only exception to this is if a user makes roughly thirty-two failed CAPTCHA attempts within a short period of time, in which case the user’s IP will be temporarily flagged. If the user’s IP is flagged they may still make a CAPTCHA attempt however both words become verification words and both must be typed perfectly. A flagged IP can be easily remedied by a resourceful user with any service that provides dynamic IP allocation or by simply using any number of proxies.



Figure

The current distortions to the words put in place by reCAPTCHA come in three layers. The whole word is skewed like a wave; it has a subtle distortion to add noise to the characters, and it has a large randomly placed color inverted ellipse. The initial step in attempting to restore the original image is to segregate every pixel in the image to ensure it is dichromatic. In the experimental results pixels within the range of 0-50 were set as white while everything else was set to black. This segregation greatly simplifies handling a grayscale image containing unused information down to a dichromatic image containing only necessary information. Figure 1 can be seen transformed with the pixel segregation in Figure 2. The next step in decoding these images would be to separate the words themselves. From here on it would be simpler to handle each word individually so a simple vertical line test is put in place to separate the words. This is accomplished by scanning along each y-axis at each x ensuring that every pixel is black at each x, if not then the line contains color and cannot be use to separate the two words.



Figure

The next significant challenge in decoding a word is removing the unsightly inverted ellipse. First the image is outlined to defeat the purpose of the random color inversion. Second the image can be pixilated to increase the speed and efficiency of the ellipse detection algorithm chosen for this system. The experimental results were created using an ellipse detection algorithm call “A New Efficient Ellipse Detection Method” (Xie and Ji, 2002). A visual representation of the pixilation along with ellipse detection can be seen in Figure 3. Of course the ellipses created for distortion by reCAPTCHA are not always perfectly symmetrical. In some cases in fact they are hardly ellipses at all, as can be seen in the second word “referendum” in each Figure. However, for the purposes of this experiment the applied detection algorithm works well enough yet a more tailored routine would most definitely yield higher results.



Figure

Now that a fairly accurate guess has been made as to the location of the inverted ellipse an algorithm can be used to remove it using basic geometry to determine whether a pixel is located inside of the ellipse. If a pixel is determined to fall within or on the boundaries of the ellipse then the pixel color is inverted, else it is left alone. After inverting back the ellipse there may be some artifacts leftover that can hinder the words character analysis, which can be easily seen at the bottom right of Figure 4 directly after an ellipse inversion, please note that the artifacts may be far more significant in number when applied to some CAPTCHAs.



Figure

These are eliminated by a few different methods. Firstly an ellipse removal routine is executed using the existing discovered ellipse parameters. This routine will examine on and inside the ellipse for white pixels along each perpendicular tangent of the ellipse. If one or two white pixels are found then they are eliminated, anymore and they are ignored. This is a very useful routine due to only examining perpendicular to the tangent along the ellipse since it greatly lowers the amount of unintentional pixel removals. Next, every white pixel is executed as a parameter of a routine that removes the white pixel if the specified area’s ratio of white pixel count over total pixel count is less than the specified rate, which along with the radius is passed as a parameter. This routine is useful in identifying “loner pixels” and ensuring they are removed prior to character analysis. Figure 5 represents a sample of Figure 4 after having been executed through these routines.



Figure

The last layer of distortion removal lies in straightening the words. This was accomplished by analyzing separately the top and bottom layer of pixels. The following description will explain the bottom portion analysis without any loss of generality applied to the top portion. The algorithm, along with the image data itself, is provided the additional parameters of the tangent width, maximum slope, and smoothing (or averaging) width. The following depicts the steps of the algorithm which will now be referred to as the “blanket algorithm”:

1) Determine the bottom most white pixels along the x-axis, store them in y\_bot, if there is not a white pixel then set y\_bot[x]=-1;

2) Start with the far left most x; x = xi = 0. Also set blanket[x]=-1

3) Set two new variables xL=x and xR=x

4) If xL-1 >= xi then xL= xL-1 or xR+1<xF then xR= xR+1; If xL >= xi goto step 7, else if xR < xF goto step 10

5) If L!=-1 && R!=-1 then blanket[x]=calc\_midPointAtX(x ,L, y\_bot[L], R, y\_bot[R])

6) If x+1<xF then x++ and loop back to step 3, else goto step 14

7) If y\_bot[xL]>-1 then tmp\_L=xL , else goto step 10

8) tmp\_m=calc\_slope(tmp\_L, y\_bot[tmp\_L], R, y\_bot[R]);

9) If tmp\_m>m && tmp\_m<max\_m then m=tmp\_m and L=tmp\_L; If xR>=xF then loop back to step 4

10) If y\_bot[xR]>-1 then tmp\_R=xR , else goto step 4

11) tmp\_m=calc\_slope(L, y\_bot[L], tmp\_R, y\_bot[tmp\_ R]);

12) If tmp\_m<m && tmp\_m<max\_m then m=tmp\_m and R=tmp\_R

13) Loop back to step 4

14) Average each blanket array value by its surrounding pixels

The algorithm checks for the best fit tangent along the bottom of the image. If a pixel is found on the right of the current x then it must generate a smaller slope than the previously found pixel on the right. If a pixel is found to the left of the current x then it must generate a larger slope than the previously found pixel to the left. Once the best fit tangent is found a simple calculation can be made to interpolate where the x would land on the tangent line. This information is stored in the blanket array and after it is smoothed lightly by averaging each integer by its neighbors it can be used to generate the red “blanket” line as seen in Figure 6.



Figure

The “blanket” algorithm works quite well for straightening words, and only faces complications when a word contains a character with a tail, such as ‘g’ or ‘y’. To modestly address this issue both the top and bottom “blanket” pixels are acquired and an average of the two determines the how far to straighten the image. Once the average has been calculated a very simple straightening routine is executed that moves each pixel down in each column along the y-axis by the difference of the “blanket” pixel and the bottom of the image. The result of such a routine can be seen in Figure 7.

After comparing Figure 1 and Figure 7 in close detail one can see how the automated cleaning of the words in the CAPTCHA can quickly and easily defeat the distortions put in place by reCAPTCHA. The entire cleaning process of the Figure 1 sample image, from pixel segregation to image straightening, took roughly 1.64 seconds on a 2.53 GHz processor. Now that the distortions are removed the words will next be analyzed for character recognition.

**5. Character Recognition**

****

Figure

One of the most widely used tools today for optical character recognition is a neural network. The research for reCAPTCHA explored various methods for implementing neural networking but in the end it was discarded for a simpler yet surprisingly more effective approach, at least in this implementation. What will be from now on referred to as a CMAP (abbreviated for Character Map) is a template composed of hundreds of different cataloged characters. To build a CMAP first a multitude of characters need to be manually segmented and labeled. This was done using a training tool that would load, remove distortions from, and display a CAPTCHA. Once the CAPTCHA was displayed the trainer can manually highlight portions of a character and then type out the word. When the user is finished training the word they can type ‘enter’ which will segment the characters, remove any unneeded space by cropping its borders, label them as described, and then catalog each of them. After training a sufficient amount of characters the CMAP is ready to be built. First the algorithm for building the CMAP will enumerate every character and take the average dimensions for every unique character to build its framework. The algorithm then cycles through every character in the catalog adding it to its respective frame in the CMAP. For instance, to add the image of the character ‘e’ to a CMAP the CMAP is first loaded and the ‘e’ frame is identified. The image is then interpolated to match the dimensions of the frame. Next, every pixel of the image of the character ‘e’ is examined and if it is white or ‘on’, then that location in the ‘e’ frame is incremented, otherwise it is not. Whenever a character is added to a frame a frame character counter is incremented. With this information it is very easy to deduce the probability that any given character will have a white or black pixel at any given point on the image. A visual representation of a lightly trained reCAPTCHA CMAP is shown in Figure 8. In Figure 8, the lighter the pixel is the more often and consequently the higher the probability that a pixel will be white when interpolated to fit that specific character’s frame. It is of no surprise that the visual representation illustrates the characters they are trained for. The CMAP generated for reCAPTCHA and for the experimental results discussed in this paper only handle the lower-case alphabet of characters (a-z). So in summary a CMAP is the combined averages of pixels for each unique character over a number of instances of that character which allows the system to easily calculate the probability that a specific point on an image will be white or black when considering it for a specific character.

One of the hardest parts of character recognition is the actual character segmentation. Especially when dealing with a CAPTCHA like reCAPTCHA where the characters are all the same color, they are distorted somewhat, and they are overlapping. It’s quite amazing how a human can glance at Figure 7 and easily recognize “presumed referendum” but when a systematic approach needs to be developed things can become quite difficult. For instance how can the computer determine that the best match for the first ‘e’ in “presumed” would be along the x-axis from 26-43 if it doesn’t know to look there? There is the option of examining every pixel range for a character match but that leads to an O() algorithm where n is the width of the image.



Figure

The approach of the system described in this paper narrows down the possible computations by running another analysis on the image using an algorithm similar to the previously described for straightening the words. With this algorithm the words are sampled for “dips” in the line found by the blanket algorithm with a few specialized parameters. The tangent width has a size of one pixel in each direction, there are no restrictions on the steep of the slopes, and there is little to no smoothing; this ensures that the “blanket line” will dip deep into the crevasses of the words as can be seen in Figure 9.



Figure

Once the “blanket” pixels have been calculated for both the top and bottom of the word with these specialized parameters a simple algorithm can be executed to determine dips in the lines. The following illustrates the steps necessary for finding “dips” on the bottom portion of the image:

1) Let x be xi+1, left = xi

2) If blanket[x]<blanket[left] left = x; else goto 4

3) x++; Loop back to 2

4) If blanket[x]>blanket[left] right = x; else goto 3

5) add\_dip((right-left)/2)

6) left = right

7) goto 3

Figure 10 depicts the detected “dips” in the form of vertical lines. With this information the total possible amount of x locations is only 20 for “presumed” and 22 for “referendum”. As opposed to 126 and 140 respectively if the “dip” algorithm was not used. This causes the character analysis to be 97.6% more efficient overall. Now that the dips are calculated they can be cycled through and compared to a previously prepared CMAP. For each dip label x1 and increment through them and for every x1 label dips to the right as x2 and increment through those. Every combination of x1 and x2 ­is segmented from the word and compared to a CMAP to determine the character with the highest probability. Once every dip combination is associated with a character and its respective probability ratio the data is ready for analysis. From here out the character ratio data calculated with this algorithm will be referred to as CRD.

One of the more interesting vulnerabilities allotted to reCAPTCHA is the requirement to have both words in the CAPTCHA be actual dictionary based words. Unlike the average CAPTCHA such as AOL’s or MSN’s where the CAPTCHA is a set of randomly generated characters without any association between them, reCAPTCHA allows a decoding system to incorporate a dictionary attack to improve efficiency significantly. The system developed for this research does indeed use a dictionary attack, although it is not entirely necessary and the algorithm to analyze the acquired character ratios could easily be modified to not require a dictionary list, it is however very useful and thus implemented. A dictionary list comprised of over 100,000 words was compiled and each word is used to analyze the CRD. A dynamic algorithm was created that returns the highest average ratio match for a word when provided the previous calculated CRD. When a word is supplied from the dictionary list the algorithm creates a list for every character in that word even if characters are redundant, then every character that matches is added to the list from the CRD. The list array will be defined as follows: list[word\_size][character\_count], there will also be another array with identical dimensions that contain newly calculated ratios and it will be defined as ratio[word\_size][character\_count]. Next, list[word\_size-1][n] gets a new ratio calculated for each letter by multiplying the width of the character by the previously calculated ratio in the CRD and this result is placed in the ratio array. This ensures that the more space that is filled up over the image by a character the higher the match will be. This is followed by analyzing list[word\_size-2][n] by checking every list[word\_size-1][i] that will allow them both to exist simultaneously on the image. If the list[word\_size-2][i] we are currently examining overlaps the list[word\_size-1][i] we are working with then it will be ignored. The list[word\_size-1][i] with the highest ratio that will accommodate both letters is added to the ratio[word\_size-2][i] along with its CRD multiplied by its character width. This logic is followed all the way down to the first letter. From here the highest ratio[0][i] is divided by the image width to get an average probability among every pixel and is returned as the result. Every word in the dictionary list is analyzed this very same way and the word with the highest probability is the final result.

**6. Experimental Results**

An unfortunate problem with calculating accurate results using reCAPTCHA is that with a large set of collected CAPTCHAs pulled from reCAPTCHA’s servers along with manually typing the correct answer is that there is no real way to be certain which of the two words is the verification word in order to determine that the CAPTCHA was decoded successfully. While a few tricks can be learned from having experimented enough with reCAPTCHA, such as if one of the words are distorted beyond a certain point, if they are shorter than a few letters such as ‘the’ or ‘to’, or if they contain numbers then it is definitely not the verification word. In some instances it is impossible to tell the difference. In an analysis of 16555 CAPTCHAs 313 of them contained a word containing a number and 6712 of them had a word of less than or equal to three characters in size. Meaning that 42.43% of CAPTCHAs supplied by reCAPTCHA can have the verification word easily determined, and that statistic is not including overly distorted words which could easily make that number far more than half. The system was setup to run for a day testing 4690 random CAPTCHAs that were not used to train the system in anyway. The results for this experiment came to 873 instances where one or both of the two words were correct. After removing all instances where the correct word was less than or equal to three characters there was a total of 630 correct. This leaves an overall efficiency of 13.4% percent. Of course there are the instances where the system is answering the digitization word correct instead of the verification word. Concluding, however, that this happens half of the time is far too conservative considering the digitization word is often more distorted, a number, or too short of a word. A safe conservative number for the results in this experiment to have accidentally answered the word other than the verification word would be 25%. Since half of the time it is not possible to guess which word is the digitization word and then there is still a 50% chance left that the solved CAPTCHA is a digitization word half of the time so .5\*.5=25%. In actual practice however, this number seems to be quite a bit too conservative as well. Unfortunately there is no environment for a larger scale real-world test against the reCAPTCHA system currently, and these figures will have to do. The rating can then be multiplied by 75% to get a highly conservative efficiency rating of 10.07%. On average the system takes roughly 20 seconds to decode an entire CAPTCHA, or 10 seconds a word. These results were executed on a single core 2.53 GHz processor. The end result for the sample CAPTCHA in Figure 1 turns out to be: ‘presume’ with a rate of 0.618, and ‘meronymy’ with a rate of 0.478.

**7. Solutions**

Unfortunately any CAPTCHA that can be read by a human can eventually be read by a computer. The only solution is to stay one step ahead of those wishing to abuse these systems by consistently changing the CAPTCHA distortions and design. While it may take the maintainer of a CAPTCHA system a couple of hours to implement a change, it takes a human no time to adjust to the difference, while a person wishing to keep their automated system working that defeats the CAPTCHA may take weeks to adopt the changes necessary to get it running again. So a very effective solution would be to create many variations of the caption distortions ahead of time and let them roll out at predetermined intervals, perhaps once every two weeks. It’s a tedious solution but it’s the only way to always keep CAPTCHA crackers one step behind. To defeat the system implemented in this research the inverted ellipse could become more random of an effect. There could be multiple inversion objects with variations of shapes. Of course there would eventually be a simple way around that method as well since outlining the image is a very simple process and greatly lowers the effectiveness of random color inversions.

**8. Conclusion**

This experiment shows that even a CAPTCHA that is considered to be highly effective and robust can be flawed and defeated. And that no system should rely entirely on their CAPTCHA system. This experiment demonstrates a few algorithms that present high efficacy when pitted against a reCAPTCHA image. The highly conservative efficiency of 10.07% is more than enough to consider reCAPTCHA defeated. Within a few attempts a system could automatically bypass the security measure and continue on with its behavior to imitate a human while having defeated a Turing test. In future works there will be better results. Unfortunately due to time constraints the training applied to the CMAP used in this experiment was not to its full potential. A half of a day’s additional training could greatly improve the end results. Also, a more effective ellipse detection routine could easily double the efficiency since the ellipses are commonly not ellipses at all. Perhaps another version of this paper will be released with results somewhere in the >50% range in the near future.

**9. Bibliography**

Xie and Ji, (2002). “A New Efficient Ellipse Detection Method” IEEE

http://www.cs.sfu.ca/~mori/research/gimpy/. “Breaking a Visual CAPTCHA”

<http://www.allspammedup.com/2010/01/google-recaptcha-cracked/> “Google reCAPTCHA Cracked”