A Simple Generic Attack on Text Captchas

Haichang Gao[1]*, Jeff Yan[2]*, Fang Cao[1], Zhengya Zhang[1], Lei Lei[1], Mengyun Tang[1],
Ping Zhang[1], Xin Zhou[1], Xuqin Wang[1] and Jiawei Li[1]
[1]. Institute of Software Engineering, Xidian University, Xi'an, Shaanxi, 710071, P.R. China
[2]. Security Lancaster & School of Computing and Communications, Lancaster University, UK
*Corresponding authors: hchgao@xidian.edu.cn, Jeff.Yan@lancaster.ac.uk

***Abstract*—Text-based Captchas have been widely deployed across the Internet to defend against undesirable or malicious bot programs. Many attacks have been proposed; these fine prior art advanced the scientific understanding of Captcha robustness, but most of them have a limited applicability. In this paper, we report a simple, low-cost but powerful attack that effectively breaks a wide range of text Captchas with distinct design features, including those deployed by Google, Microsoft, Yahoo!, Amazon and other Internet giants. For all the schemes, our attack achieved a success rate ranging from 5% to 77%, and achieved an average speed of solving a puzzle in less than 15 seconds on a standard desktop computer (with a 3.3GHz Intel Core i3 CPU and 2 GB RAM). This is to date the simplest generic attack on text Captchas. Our attack is based on Log-Gabor filters; a famed application of Gabor filters in computer security is John Daugman's iris recognition algorithm. Our work is the first to apply Gabor filters for breaking Captchas.**

I. Introduction

Captcha allows websites to automatically distinguish computers from humans. This technology, in particular text-based Captchas, has been widely deployed on the Internet to curb abuses introduced by automated computer programs masquerading as human beings. Although many text Captchas have been broken, the most recent studies, such as one by a UC Berkeley team [21] and one by Stanford and Google [6], suggest that Captchas are still an effective security tool.

Captcha has had many failure modes. Designers typically learn from previous failures to design better schemes. Current Captchas are much more sophisticated than the earliest generation designed at Carnegie Mellon. As predicated in [25], this technology has been going through a process of evolutionary development, like cryptography, digital watermarking and the like, with an iterative process in which successful attacks lead to the development of more robust systems.

The robustness of text Captchas has been an active field in the research communities. Many attacks have been proposed. For examples, in 2003, Mori and Malik used sophisticated object recognition algorithms to break two early designs: EZ-Gimpy and GIMPY [18]. In 2005, Chellapilla and Simard attacked many early Captchas deployed on the Internet [19]. Yan and El Ahmad broke most visual schemes provided at Captchaservice.org in 2006 [24], published a segmentation attack on Captchas deployed by Microsoft and Yahoo! [25] in 2008, and broke the Megaupload scheme with a method of identifying and merging character components in 2010 [1]. In 2011, Bursztein et al. showed that 13 Captchas on popular websites were vulnerable to automated attacks, but they achieved zero success on harder schemes such as reCAPTCHA and Google's own scheme [5]. In the same year, Yan's team published an effective attack on both of these schemes [2]. At CCS'13, Gao's team and Yan jointly published a successful attack on a family of hollow schemes [13]. The latest attack on Captchas [4] was published in August 2014.

As a side note, other notable attacks include [14, 17, 20, 23, 27]. But they studied alternative Captcha designs such as animation, image and audio schemes, rather than text ones. Therefore, we will not look into the details.

These fine prior art advanced the scientific understanding of Captcha robustness, but most of them have a limited applicability. Many of them broke specific schemes, and only a few broke a security mechanism as a whole. We quote the following from a well-cited paper [25].

> The relatively wide applicability of our attack on the MSN scheme is encouraging. However, we doubt that there is a universal segmentation attack that is applicable to all text Captchas, given that hundreds of design variations exist. Instead, a more realistically expectation is to create a toolbox (i.e. a collection of algorithms and attacks, ideally organised in a composable way) for evaluating the strength of Captchas.

This toolbox approach has been a common practice (with a few exceptions) in the Captcha research community, as evidenced by papers published afterwards. Decaptcha [5], was a well conceived tool for analysing Captcha robustness and was considered to be a generic attack, but it followed such a toolbox approach, as we will explain in details later.

In this paper, we propose a simple but effective attack that breaks a wide range of text Captchas. Our attack is based on Log-Gabor filters, a versatile signal processing technique. A key innovation of John Daugman's iris recognition algorithm was to encode iris patterns into binary bits using 2D Gabor filters [10]. Our attack uses 2D Log-Gabor, a variant of Gabor filters. By convolving a Captcha image with Log-Gabor filters of four different orientations (i.e. directions) respectively, we

NDSS '16, 21-24 February 2016, San Diego, CA, USA
 ISBN 1-891562-41-X
http://dx.doi.org/10.14722/ndss.2016.23154

extract character components along each orientation. Then, we use a recognition engine to combine adjacent components in different ways to form individual characters. The most likely combination is output as our recognition result.

We have tested our attack on Captchas deployed by top 20 most popular websites according to Alexa ranking [3]. These real-world Captchas include Google's new reCAPTCHA, hollow schemes, and conventional designs; they represent a wide range of design features. We have also tested our attack on much harder Captchas such as an old version of reCAPTCHA and two other designs. Our attack is designed to aim for simplicity and general applicability, rather than high success rates for breaking individual schemes. However, it has successfully broken all the schemes we tested, judged by both criteria commonly used in the Captcha community [5, 7]. For most of the schemes, it has achieved a good success rate.

Novelty and significance. Our attack uses a single segmentation and recognition strategy, and it is to date the best in terms of simplicity, power and general applicability. Breaking some Captchas is rarely news, but breaking all the Captchas with a single method that is so simple is surprising (even to ourselves). Although we have had much experience in breaking various Captchas, we did not expect at the beginning that our method would work so well.

Our attack might suggest that the current common practice of text Captcha designs is doomed, but it does not pronounce a death sentence to the idea of text Captcha altogether. It's highly likely that new text Captchas will be invented. We are experimenting some new ideas, for example.

On the other hand, another important value of our attack is that it can be used as a standard test: any new design that cannot pass this test should not be deployed. Moreover, for people working in security economics, this work also suggests the possibility that adversaries can launch concerted automated attacks on Captchas to reduce their cost.

We organise this paper as follows. Section II briefly introduces the essence of Log-Gabor filters. Section III describes popular real-world Captchas we collected from top 20 web sites. Section IV presents technical details of our attack. Section V evaluates our attack empirically and compares it with prior art. Section VI examines various design alternatives and shows that our attack is optimal among these design choices. In Section VII, we argue that common countermeasures only provide a partial defence against our attack. Section VIII discusses our attack's implications and concludes the paper.

II. Gabor filters

Gabor filters are powerful signal processing algorithms, and they offer the best localization of spatial and frequency information simultaneously. Nobel Physics Prize winner Dennis Gabor laid their theoretical foundations in 1946. A complex Gabor filter is defined as the product of a Gaussian kernel and a complex sinusoid. The temporal (1-D) Gabor filter can serve as excellent band-pass filters for unidimensional signals (e.g., speech). John Daugman extended Gabor's work to invented the Spatial (2-D) Gabor Filter [9].

Gabor filters have two main limitations. The maximum bandwidth of a Gabor filter is limited, approximately about one octave. If the bandwidth is larger, a non-zero DC component will exist. If a wide spectrum is needed, Gabor filters are not optimal.

Proposed by David Field in 1987, Log-Gabor filters [11] improve normal Gabor filters in the following sense. Log-Gabor's transfer function is a Gaussian on a logarithmic frequency axis. Normal Gabor filters often over-represent the low frequencies, but it is not the case for the Log-Gabor. Log-Gabor filters allow arbitrary bandwidth and the bandwidth can be optimised to produce a filter with minimal spatial extent. Field suggested that Log-Gabor filters encode natural images more efficiently than ordinary Gabor functions, and that the former are consistent with measurements of mammalian visual systems which indicate we human beings have cell responses that are symmetric on the log frequency scale.

Mathematically, 2D Log-Gabor filters are constructed in the polar coordinate system of frequency domain as follows.

$$G(f,\theta) = G(f)\cdot G(\theta) \tag{1}$$

$$G(f) = \exp\{-[\log(f/f_0)]^2/[\log(\sigma/f_0)]^2\} \tag{2}$$

$$G(\theta) = \exp[-(\theta-\theta_0)^2/2\sigma_\theta^2] \tag{3}$$

f and θ represent the radial and angle coordinate, respectively. f_0 and θ_0 represent center frequency and direction of the filter, respectively. σ and σ_θ represent radial bandwidth and directional bandwidth of the filter.

$G(f)$ is the radial component that controls the bandwidth of the filter, and $G(\theta)$ is the angle component that controls the choice of filter orientations. $G(f,\theta)$ defines a complete 2D Log-Gabor function. By definition, Log-Gabor filters always have no DC component.

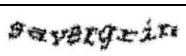
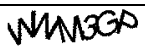
Gabor filters were used before in the context of computer security, but mainly in the field of biometrics. The most famous application of Gabor filters in computer security is Daugman's iris recognition [10]. Our work is the first application of Gabor filters to analyse Captcha robustness. A study [8] proposed to construct Captchas using Gabor sub-space, but its contribution is entirely orthogonal to ours.

III. Real World Popular Captchas

We aim to use a wide range of real-world Captchas, each with distinct design features, to evaluate the effectiveness of our attack. We choose those used by the top 20 most popular web sites (including Google, Facebook, Youtube, Linkedin, Twitter, Blogspot, Wordpress, Yahoo!, Baidu, Hao123, Wikipedia, QQ, Microsoft, Amazon, Taobao, Sina and Ebay), since they all use popular text-based Captchas. Some of the websites use the same Captcha scheme. For example, Google, Youtube, Facebook, Linkedin, Blogspot, Wordpress and Twitter all use reCAPTCHA. We have collected in total 10 Captcha schemes, as summarized in Table I. With regard to the reCAPTCHA scheme, we are interested only in control words, i.e. the right part of each challenge. The left part is not a text scheme, but involves with a different image recognition task.

According to font styles and positional relationships between adjacent characters, current text-based Captchas can

TABLE I. TARGET CAPTCHA SCHEMES.

Scheme	Website	Sample Captcha	Characteristics
reCAPTCHA	google, facebook, youtube, linkedin, twitter, blogspot, wordpress, google.co.in		CCT scheme, only digits used, rotation used, varied font sizes, varied Captcha lengths.
Yahoo!	yahoo.com, yahoo.co.jp		hollow scheme, varied fonts, rotation and distortion used, varied Captcha lengths
Baidu	baidu.com hao123.com		CCT scheme, rotation used
Wikipedia	wikipedia.org		Character isolated scheme, varied Captcha lengths, no digits used
QQ	qq.com		Hollow scheme, rotation used, overlap used, varied font sizes
Microsoft	live.com bing.com		Character isolated scheme, varied Captcha lengths, varied font sizes, rotation used
Amazon	amazon.com		CCT scheme, constant font, rotation used
Taobao	taobao.com		CCT scheme, rotation used, large alphabet set
Sina	sina.com.cn		CCT scheme, background clutter, noise arcs used
Ebay	ebay.com		CCT scheme, varied font sizes, rotation used

be classified into three categories: character isolated schemes, hollow character schemes and 'crowding characters together' (CCT) schemes. Clearly, our target schemes cover all these categories. For example, there are character isolated schemes (e.g. Microsoft and Wikipedia), hollow schemes (e.g. Yahoo! and QQ) and CCT schemes (e.g. reCAPTCHA and Baidu).

Moreover, some schemes are with noise arcs (e.g. Sina), but some without (e.g. Taobao and Ebay). Some schemes use a fixed string length (e.g. Amazon and Taobao), but some with a varied string length (e.g. reCAPTCHA and Yahoo!). Some schemes use rotation, and some do not. Fonts used vary across different schemes, too.

Overall, these schemes represent a wide spectrum of designs, each with distinctive features.

IV. OUR ATTACK

Our attack includes two main steps:

1) Extracting components. Log-Gabor filters are used to extract character components from Captcha images along four directions, respectively. In contrast to previous attacks such as [4, 13], preprocessing is unnecessary for our attack, and Log-Gabor filters are applied directly to the images.

2) Partition and recognition. A recognition engine is used to try different combinations of adjacent components, and then the most likely combination (or partition) is chosen as the correct recognition result. We choose k-Nearest Neighbours (KNN) as our recognition engine, because KNN is a top performer in text recognition [16].

In the following, we explain the detail of our attack, using Microsoft, QQ and Baidu Captchas as examples. They are representatives of the three design categories, namely character isolated schemes, hollow character schemes and CCT schemes.

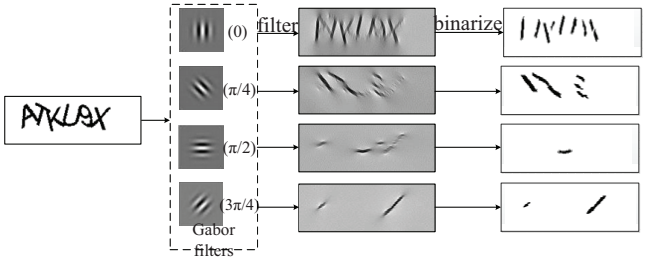


Fig. 1. Extracting character components.

A. Extracting Components

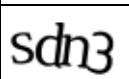
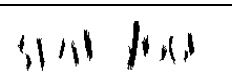
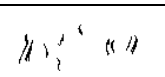
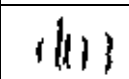
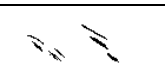
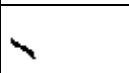
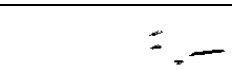
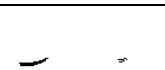
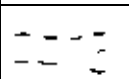
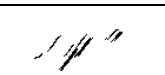
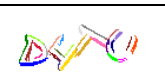
This step uses Log-Gabor filters to extract character information, as shown in Figure 1. We set θ to four different angles, 0, $\pi/4$, $\pi/2$, and $3\pi/4$. That is, we extract character information along the four directions by convolving a Captcha image with each of the filters respectively. We set f_0 to 1.414, an empirical setting that makes extracted components clearly visible. We set σ_θ to $\pi/8$, σ/f_0 to 0.55, resulting in a bandwidth of roughly 2 octaves, which achieve a good balance between retaining texture structure and removing noise. These configurations remain the same for all our target Captchas.

This filtering operation is directly applied to gray-scale images, and then the resulting images are binarised to get character components in black and white.

Table II shows for each of the schemes our extraction result along each of the four directions. Each character component is extracted out along the direction that is closest to it. Among

the four directions, it is possible that no component is extracted at all at some directions, but this is not an issue of concern. In fact, we discard small components extracted, with little impact on our follow-up recognition. For the purpose of illustration, Table II also shows a superposition of character components extracted from all four orientations.

TABLE II. EXTRACTION RESULTS.

	Microsoft	QQ	Baidu
Angle			
0			
$\pi/4$			
$\pi/2$			
$3\pi/4$			
+			

Note: in this paper, extracted character components are shown in different colors so that readers can easily distinguish them from each other.

B. Partition and Recognition

After extracting components, we try to find the most likely correct combination of adjacent components to form individual characters. Typically, the number of components is larger than the number of characters to be formed. Therefore, there will be many possible combinations (or partitions). We use a systematic and efficient algorithm to achieve partition and recognition simultaneously as follows. (Due to page limit, the Baidu scheme is used to explain key techniques in this step, but key details of attacking Microsoft and QQ schemes are shown in Appendix.)

Step 1. Component sorting. Extracted components are stored in no more than four separate images of the same dimension. We apply Color Filling Segmentation (CFS) [25] to pick up all the components from each image, and we record the coordinates (x, y) of each component's top-left pixel. All the components are then sorted by these coordinates, and the rules for ranking order are the following: x-coordinate has a higher priority than y-coordinate; the smaller x-coordinate (i.e. more left), the higher rank; the smaller y-coordinate (i.e. more upper), the higher rank. The sorted components are then numerically ordered, starting with 1 meaning the highest rank.

Figure 2 shows an example, where component 1 has the leftmost pixel among all components and thus is rank-ordered as number 1; and component 11 has the leftmost pixel among components 11 to 13.

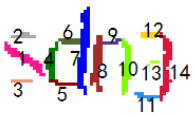


Fig. 2. All components rank ordered.

Effectively, this step is like creating a superposition of four extracted images, and then sorting all the extracted components in a particular order via the above algorithm.

Step 2. Graph building. Our algorithm constructs an $n \times n$ table, where n is the total number of components. For the example in Figure 2, $n = 14$.

A cell (i, k) at the intersection of row i and column k in the table indicates whether it is feasible to combine components i, i+1,$\cdots$, k all together to form a larger single component. If such a combination is feasible, the cell (i, k) will be marked with '•'. Otherwise, the cell (i, k) will be set to NULL. The infeasible case occurs only in one of the following scenarios: (1) when i is larger than k (i.e. when a cell's row index is larger than its column index, which should be omitted, since we combine components only in a monotonically increasing order); or the combination is either (2) too wide or (3) too thin to form a legitimate character. (Note: the largest possible character width and the smallest possible character width can be empirically established with a simple analysis of a sample dataset; this is a trivial task.)

The initial table for the example in Figure 2 is shown in Table III, where all plausible component combinations are marked by '•'.

The $n \times n$ table gives all the plausible component combinations for an image. Our ultimate task is to use information in the table to find the most likely way of forming characters, i.e., finding the best partition. This table is effectively a graph. Figure 3 gives a directed graph that is equivalent to Table III.

TABLE III. THE INITIAL $n \times n$ TABLE FOR THE EXAMPLE IN FIGURE 2.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	•	•	•	•										
2				•										
3				•										
4					•	•	•	•						
5							•	•						
6								•						
7									•	•				
8										•				
9											•	•	•	
10											•	•	•	•
11														•
12...14														

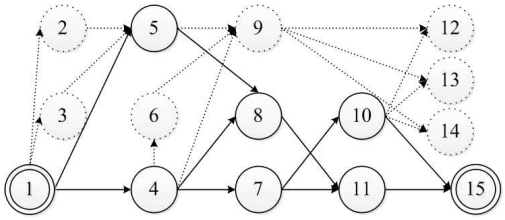


Fig. 3. The equivalent graph of Table 3.

This graph building process is similar to the method in [13]. However, a main difference is that they call the recognition engine to produce a recognition result for each plausible combination, but we do not call the engine at all.

Step 3. Graph pruning. A node on a graph can be redundant for our purpose, if there is no feasible path among all those passing through this node. We use the following algorithm to detect and remove any redundant node:

i) For each node i ($i \neq 1$ and $i \neq n+1$), using Dijkstra algorithm to compute the shortest path from node 1 to node i, and the shortest path from node i to node n+1;

ii) If the sum of the length of these two shortest paths is larger than the largest possible Captcha string length, node i will be removed as a redundant node; its connecting edges will be removed, too. The rationale is simple: the length of a valid path from node 1 to n+1 should not be larger than the number of characters in a Captcha string;

iii) If there is no path from node 1 to node i, or no path from node i to node n+1, we set the length of the corresponding shortest path to infinity;

iv) This process repeats recursively until no further nodes are removed after a traversal.

Redundant nodes and their connecting edges in Figure 3, as detected by the above algorithm, are marked with dotted lines, indicating that they are to be removed.

Step 4. Recognising component combinations. Then a trained KNN is used to determine which character each of the remaining edges in the graph is likely to be. (Preparing KNN is straightforward, and explained in Section V). We then update cell (i, k) in the table with the recognition result returned by the KNN engine for a corresponding edge.

TABLE IV. THE FINAL $n \times n$ TABLE GENERATED BY KNN.

	1	2	3	4	5	6	7	8	9	10	11...13	14
1			***s/0.81***	s/0.52								
2,3												
4						c/0.75	***d/0.87***					
5							d/0.68					
6												
7									k/0.44	b/0.43		
8										***n/0.80***		
9												
10												3/0.58
11												***3/0.84***
12...14												

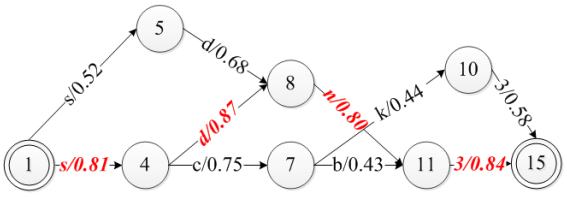


Fig. 4. The equivalent graph of Table 4.

Table IV shows the updated $n \times n$ table, and its equivalent graph is shown in Figure 4. For example, both the cell (1, 3) in the table and the edge from node 1 to node 4 in the graph indicate that KNN recognises the combination of components 1 to 3 as 's' with a confidence level of 0.81.

Step 5. Graph search. Now we search the graph to find an optimal partition. We adopt a dynamic programming (DP) approach for our graph search, which will find the optimal partition in only one traversal.

We define that the target problem of DP is to select the path ending at node n+1 with the largest confidence value sum and the corresponding step (i.e. the number of edges on the path) is equal to the Captcha string length (i.e. the number of characters). Note: this does *not* mean that our algorithm is applicable only to Captchas with a fixed string length. Instead, we easily handle those with a varied string length, e.g. by enumerating all possible lengths (typically from 4 to 12), with little performance penalty.

The overlapping sub-problem for DP is for each node j, the confidence-level sum along the path ending at j should be the largest. Note that for a path ending at node j, there may be several possible edge numbers and the largest confidence-level sum of each case should be recorded, as illustrated in Table XV in Appendix. The sub-problem's solution is worked out with a bottom-up approach, i.e., the solution of node j is worked out by that of its precursor.

The following pseudo code illustrates our DP process. The traversal starts from node 1, and ends at node n+1; the nodes are traversed in an ascending order. An array $value$ stores the confidence-level sum of each possible step for each node, $result$ stores the corresponding result string for each node, $step$ stores the number of current recognised characters. R is the final recognition result and v is its corresponding confidence level sum, $confidence$ and $recochar$ represent the recognition confidence level and the result of each feasible component combination, respectively. For example, $confidence[i, j]$ is the confidence level calculated by KNN for the combination formed by combining components from i to j.

Function $GetValue(j)$ works out the largest confidence sum and the corresponding result of each $step[j]$ for node j (i.e. $value[j]$, $result[j]$), in which $prej$ is a list that stores all the precursors of node j. Function $Main$ works out the $value[n+1]$ and $result[n+1]$. This is a bottom-up process since we calculate from $value[1]$ to $value[n+1]$ in sequence. The final recognition result R is got by Function $Select(num)$.

Generated by our attack program, Table V shows with the example in Figure 2 the process of finding the optimal partition with our DP algorithm. DP simplifies the search process by recording the largest confidence sum of each node. The italic item highlighted in the table indicates the optimal partition that has the highest confidence-level sum. That is, "*sdn3*" is the recognition result in this case.

Procedure Main()
Begin
$R \leftarrow NIL$
$v \leftarrow 0$
for $j \leftarrow 1$ *to* $n+1$
$value[j] \leftarrow 0$
$result[j] \leftarrow null$
$step[j] \leftarrow 0$
if $j > 1$
$GetValue(j)$
$Select(n+1)$
End
Procedure GetValue(j)
Begin
foreach i *in* $prej$
if $value[i] + confidence[i,j] > value[j]$
$value[j] \leftarrow value[i] + confidence[i,j]$
$result[j] \leftarrow$ **strcat**$(result[i], recochar[i,j])$
$step[j] \leftarrow step[i] + 1$
End
Procedure Select(num)
Begin
foreach i *in* $step[num]$
if i *in* $Captcha\ length$
if $value[num] > v$
$v \leftarrow value[num]$
$R \leftarrow result[num]$
End

TABLE V. THE SEARCH PROCESS.

j	$step[j]$	Path	$value[j]$	$result[j]$
4	1	1→4	0.81	s
5	1	1→5	0.52	s
7	2	1→4→7	1.56	sc
8	2	1→4→8	1.68	sd
10	3	1→4→7→10	2.00	sck
11	3	1→4→8→11	2.48	sdn
15	***4***	***1→4→8→11→15***	***3.32***	***sdn3***

V. EVALUATION

A. Attack Results

We have implemented our attack in C# and tested it on all the target schemes on a desktop computer with a 3.3GHz Intel Core i3 CPU and 2 GB RAM. We follow common practices in the literature to evaluate our attack.

Data Collection. For each scheme in Table II, we collected from the corresponding website 500 random Captchas as a sample set, and another 500 as a test set. The choice of target schemes follows a single and objective criterion: their popularity by Alexa ranking. We collected all the data randomly, and our data collection was carried out during 2013-2015.

In this period, the schemes we study are relatively stable, except that reCaptcha has adopted a non-text scheme.

KNN Engine. Character samples we extracted from sample sets are all normalized to 28*28 pixels. KNN is a simple and effective classifier in text recognition. To do character recognition, we measure the similarity between corresponding pixels of two images. The confidence level of a recognition result is also derived from this similarity value.

We assign a larger weight to similar black pixels, but a smaller weight to similar white pixels, in order to decrease the importance of matching background pixels in decision making. If two corresponding pixels do not match, a negative value will be added to the similarity calculation.

The recognition rate achieved by KNN depends on both the sample size and the value of k. We determine k value via cross-validation.

Success rate. Our attack's success rate and average speed on each scheme are summarized in Table VI. Our success rates range from 5.0% to 77.2%, and for a majority of the schemes, the minimum success rate is 16.2%.

TABLE VI. ATTACK RESULTS.

Scheme	Success rate	Speed(s)
reCAPTCHA	77.2%	10.27
Yahoo!	5.0%	28.56
Baidu	44.2%	2.81
Wikipedia	23.8%	3.74
QQ	56.0%	4.95
Microsoft	16.2%	12.59
Amazon	25.8%	13.18
Taobao	23.4%	4.64
Sina	9.4%	4.83
Ebay	58.8%	5.98

A commonly accepted goal for Captcha robustness is to prevent automated attacks from achieving higher than 0.01% success [7]. But this goal was considered too ambitious by some researchers. For example, [5] suggested that a Captcha scheme is broken, if an automated attack achieves a success rate of 1%. According to either criterion, our attack has broken all the Captchas deployed by the top 20 websites.

Our success rates on Yahoo! and Sina are relatively low. For the Yahoo! scheme, our extraction method breaks a long text string into a large number of (tiny) components, which produces a huge possible set of combinations. The warping and overlapping mechanisms used in this Captcha turns out to be disruptive to our component sorting algorithm, making our recognition less successful.

For the Sina scheme, because noise arcs are similar to character components, and thus extracted out by our directional filtering – they interfered our recognition engine. Those intersecting arcs that cut through characters are particular troublemakers.

For the sake of generality and simplicity, no ad-hoc processing is used in our attack. It is unsurprising that appropriate preprocessing can improve our attack's performance – for example, in our experiments, some simple hollow filling and noise arc removal boosted our success rates on Yahoo and Sina schemes to 10.0% and 21.0%, respectively. Probably more important, it is worthy noting that without any preprocessing or scheme-specific optimisations, our attack works on all the schemes, and thus demonstrates robustness to hollow fonts and noise arcs to some extent.

Speed. On average, it takes 3 to 14 seconds for our attack to break most of the schemes. The slowest speed was on the Yahoo! scheme, nearly 29 seconds – still acceptable, as it is an excessive usability requirement to demand every human user to solve a Captcha in less than 30 seconds; some Captchas

deployed in the wild reported an average solving speed of more than 46 seconds [17].

The following reasons explain that it takes more time to attack the Yahoo! scheme than others. First, it used a much longer text string than other schemes. Second, because it is a hollow scheme, our extraction method breaks the whole string into a large number of components (see Figure 10(a) for an illustration). This slows down our recognition speed significantly. Third, it used digits, upper and lower case letters, and thus had a relatively large alphabet set. This means it takes more time for the engine to do comparison and recognition.

The fastest speed was on the Baidu scheme. In this scheme, only four characters are used in each challenge. Thus the extraction process produced much less character components than with other schemes, and this significantly reduces our attack time.

Clearly, our attack is efficient and poses a realistic threat to all these schemes.

B. Further Applicability Test

We test our attack on the following Captchas that are generally considered hard.

An old version of reCAPTCHA (Figure 5). The Stanford team achieved a zero success on attacking this scheme, as reported in CCS'11 [5]. The reCAPTCHA version that we broke in the previous section is the new version, which was carefully tuned and rolled out by Google in September 2013, as reported by its designers in [6].

(a) Original image (b) Reconstruction

Fig. 5. Early reCAPTCHA.

(a) Original image (b) Reconstruction

Fig. 6. Yandex Captcha.

Yandex scheme (Figure 6). As the largest Russian search engine in the world, Yandex uses its Captcha in user password recovery. This is a hollow Captcha, and has never been broken in the literature. Gao et al's attack [13] successfully broke a number of hollow Captchas, but it was not tested on the Yandex scheme. We implemented their attack, but it failed to break the Yandex scheme in our experiments, for the following reasons. Broken contours are heavily used in this design, and so are thick intersecting interference arcs (i.e., those that cut through characters). Both are defence methods recommended by [13] to defeat their attack; these mechanisms make it hard to extract character strokes from hollow Captchas.

In contrast, our attack reported in this paper achieves a success rate of 7.8% on reCAPTCHA and 2.2% on the Yandex scheme. The average attack speed is 8.06 and 15.5 seconds, respectively. Our attack achieved a lower success rate on the older reCAPTCHA than on its current version; but the latter has much better usability, as shown in [6]. The older reCAPTCHA is rarely used now, probably due to its usability concerns.

The most recent work by Google [15] achieved a much higher success for attacking the old reCAPTCHA version than we do. However, they used millions of training samples, whereas we used only 500. Also, their approach requires sophisticated deep-learning algorithms, advanced distributed computing infrastructure, and computers with powerful CPUs and huge memory. Moreover, it is unclear how effectively their approach will work on other Captchas.

We also test our attack on a hard Yahoo! scheme (see Figure 7), which was the hardest among all the schemes broken by [4]. Our attack achieved a success rate of 9.2%, better than the result (5.33%) reported in [4]. Note: as will be compared later, our attack is also much simpler than theirs.

C. A Comparison with Prior Art

The series of works by Yan and El Ahmad [2, 24, 25] lead to methods like pixel counting, histogram analysis, and CFS. These methods are often used as building blocks in successful attacks, but when used alone, only occasionally constitute a successful attack.

Decaptcha, proposed in [5], claimed to be a generic attack, and it works as follows. Decaptcha uses a five-stage pipeline: preprocessing, segmentation, post-segmentation, recognition, and post-preprocessing. In each stage, various techniques were used for different Captchas. For example, in preprocessing stage, algorithms such as anti-pattern methods and Markov Random Field algorithm are used to de-noise a Captcha. In the most critical segmentation stage, Decaptcha 'attempts to segment the Captchas using various segmentation techniques, the most common being CFS which uses a paint bucket flood filling algorithm' [5]. Combining a variety of algorithms and methods as "lego bricks" is a key feature of Decaptcha – it follows the very toolbox approach. On the other hand, Decaptcha failed to break the early reCAPTCHA, whereas our attack can break it. The attacks implemented by Decaptcha cannot break hollow Captchas, either; but ours can.

In December 2013, a startup company Vicarious [22] claimed in a video that they designed a method to break a number of Captchas. Since they revealed no technical details, it is impossible to determine their work's validity, and impossible to judge whether their method is similar to ours or how it differs. Also, they claimed success only on reCAPTCHA, Yahoo!, Paypal and several (very simple) Botdetect schemes. Our target schemes are a much wider range and more representative collection of high-profile Captchas.

Gao et al's attack on a family of hollow Captchas [13] is the first work of solving Captchas in a single step that uses machine learning to attack the segmentation and the recognition problems simultaneously. They first extract character components from hollow fonts, and then try various combinations with a recognition engine. However, their method only works on hollow Captchas, as their success in separating connected

characters vitally relies on intrinsic properties of hollow fonts. Their method **cannot** separate non-hollow characters that connect with each other, and thus **cannot** break non-hollow Captchas. Moreover, even for hollow schemes, their method requires extensive and sophisticated pre-processing, whereas our attack does *not* require any traditional pre-processing except binarisation, a trivial process that converts an image from color or gray-scale to black and white. Their recognition method is similar to ours, but our graph search algorithm is significantly improved, compared to theirs (a detailed comparison is in Section VI). Overall, our attack is much simpler than theirs, but with a much wider applicability, e.g. working on both hollow and non-hollow Captchas. Note: among all the 10 schemes our attack has broken in this paper, only two of them (Yahoo! and QQ, both hollow schemes) can be broken by the attack proposed in [13].

Bursztein et al [4] is the second attack that addresses segmentation and recognition simultaneously, and it has broken multiple Captchas. This attack analyses all possible ways of segmenting a Captcha, and thus it is a brute-force approach in essence. It works as follows.

As illustrated in Figure 7, they first scan the top pixels of the Captcha to generate a curve, and scan the bottom pixels to generate another curve. Then they identify inflection points by examining the second derivative of the curves. Each potential cut or segment is constructed by connecting the inflection points - one from the top, and one from the bottom. This method produces an exponential number of segments or cuts.

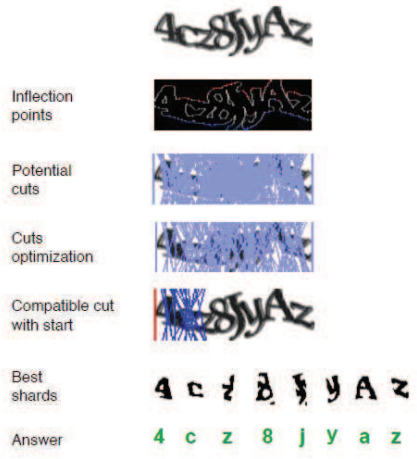


Fig. 7. The attack introduced in [4].

Then they use various heuristics to reduce the number of cuts, like removing all the cuts that have an angle larger than 30°, examining the ratio of white pixels to black ones to eliminate cut lines that pass through too many black pixels, comparing pixel intensities of the left and right boundaries to estimate a transition between two letters, and finding cuts compatible with starting positions.

Next, they use a classifier to pick the "best shards" among the remaining segments by manually assigning higher weight to pixels near the centre of the segment, and to darker pixels. Finally, ensemble learning is used to identify among each sequence of segments the best possible one as the result.

As the computational cost of their attack "increases exponentially with the length of the Captcha, to the point of becoming prohibitive on long Captchas", they also resort to various optimisation strategies to tweak recognition algorithms, e.g. by considering a window of two letters at a time, to improve the trade-off between speed and accuracy. To improve recognition results, they also apply reinforcement learning, i.e. asking human to manually identify and annotate segments that have been misclassified.

Their attack is significantly more complex than ours, and we do not need any of the heuristics they used, as well as the human efforts they relied on.

VI. Design Choices

In this section, we discuss various design alternatives, and show that our attack is optimal among these design choices.

A. Graph Search Algorithms

We first compare our graph search algorithm (Section 4) with two related algorithms.

Gao et al. algorithm [13] is based on Depth-First-Search (DFS). It starts from node 1 in the graph and explores along each branch until the path length reaches the Captcha string length before backtracking. All paths of a length equaling to the Captcha string length in a graph are traversed using DFS, and then the path ending at n+1 with the largest confidence level sum is selected. This DFS algorithm is not optimal, since it explores paths that can't reach the right edge of the graph, and re-explores previously visited nodes after their best following partition has been discovered.

Integer partition algorithm is another novel graph search algorithm that we conceive for our attack. The rationale is the following. Assume that m is the Captcha length, our task is to find the most likely way of forming m characters using n components, i.e., finding the best partition. This task is similar to the classical 'integer partition' problem: in number theory and combinatorics, a partition of a positive integer n, is a way of representing n as a sum of m positive integers. We first work out all partitions that divide integer n into m parts, then select the partition with the largest confidence sum.

Compared with the DFS graph search, this algorithm reduces the search space by skipping paths that do not end at node n+1. However, it requires working out all partitions that divide n into m parts, which is expensive.

Our new algorithm introduced in Section 4 is optimized, compared with both methods discussed above, for the following reasons.

First, it prunes the graph to remove all redundant nodes, and thus reduces the number of times we call the recognition engine, and reduces the time consumption of our attack. As it takes about 0.04 seconds to execute a single call to the KNN in our experiment, if many possible combinations require calling the KNN, the recognition time in total will significantly increase. On the other hand, after our graph pruning, sometimes there remains only a single path, which is exactly the optimal partition that we look for. Figure 11 shows such an example.

Second, our graph search adopts a dynamic programming approach. It finds the optimal partition in only one traversal, preventing re-exploring visited nodes.

Empirical evaluation. We implemented all the three algorithms, and compared the results of our attack facilitated by different search algorithms. Note: all the three algorithms can handle Captcha schemes with a varied length.

With different search algorithms, our attack achieved the same success rate. That is to say, the choice of search algorithms does not have an effect on our attack's success rate. However, as shown in Table VII, the integer partition algorithm improves the attack speed achieved by the old search algorithm for each Captcha scheme. Our new DP search algorithm further improves the attack speed significantly; the figures in Table VII include the time for graph pruning, and therefore this is a fair comparison.

To sum up, both theoretical and empirical analyses in the above suggest that our new graph search algorithm outperforms both alternatives.

TABLE VII. ATTACK SPEED VS. GRAPH SEARCH ALGORITHMS.

Scheme	Average attack speed (Seconds)		
	DP search	Integer partition algorithm	DFS search
reCAPTCHA	10.27	10.31	10.87
Yahoo!	28.56	33.33	34.32
Baidu	2.81	3.00	3.14
Wikipedia	3.74	3.78	3.83
QQ	4.95	5.15	5.55
Microsoft	12.59	14.93	15.49
Amazon	13.18	14.60	15.28
Taobao	4.64	4.74	4.80
Sina	4.83	4.93	5.03
Ebay	5.98	6.01	6.06

B. Extraction Orientations

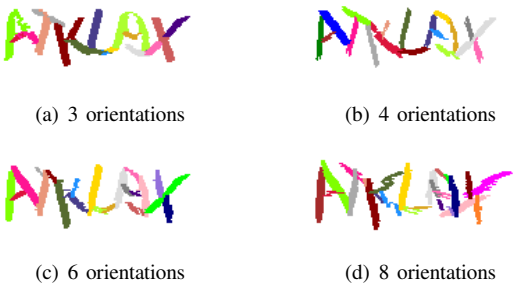
We tested our Gabor filters with different combinations of extraction directions:

3 orientations: 0, $\pi/3$, $2\pi/3$;

4 orientations: 0, $\pi/4$, $2\pi/4$, $3\pi/4$;

6 orientations: 0, $\pi/6$, $2\pi/6$, $3\pi/6$, $4\pi/6$, $5\pi/6$;

8 orientations: 0, $\pi/8$, $2\pi/8$, $3\pi/8$, $4\pi/8$, $5\pi/8$, $6\pi/8$, $7\pi/8$.



(a) 3 orientations (b) 4 orientations

(c) 6 orientations (d) 8 orientations

Fig. 8. Superimposition of extracted components.

Figure 8 shows a superimposition of the extraction results achieved by each configuration. Judged by the superimposition quality, Gabor filters with 4 directions achieves the best performance. When fewer orientations are used, character pixels along some directions will be missing. When more orientations are used, character components become fragmented, and the increased number of components will decrease our attack's speed and success rate. These are confirmed by our empirical results as shown in Table VIII.

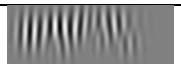
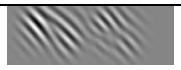
TABLE VIII. ATTACK RESULTS ON AMAZON CAPTCHA WITH DIFFERENT ORIENTATION CONFIGURATIONS

Orientations	Success rate	Average attack speed (Seconds)
3	20.8%	12.25
4	25.8%	14.32
6	9.2%	21.55
8	7.4%	30.01

C. Extracting Methods

2D Gabor filters [10] and steerable filter banks [12] can extract texture features from an image at any direction. We tested both for extracting character components in Captcha images. We compared them with Log-Gabor filters in Table IX. Log-Gabor filters are the best for our purpose.

TABLE IX. A COMPARISON OF DIFFERENT FILTERS.

	2D Gabor	Steerable filter	Log-Gabor
0			
$\pi/4$			
$\pi/2$			
$3\pi/4$			

D. Classifiers

We tested Support Vector Machine, Back-Propagation Neural Network, Template Matching and Convolutional Neural Network (CNN, a multi-layer neural network doing deep learning and extracting features from training samples automatically and efficiently) as a candidate for our recognition engine. Among these classifiers, CNN achieved the fastest attack speed and the best success rate. This result is consistent with the comparison in [13].

We also compared the performance of KNN and CNN. As shown in Table X, KNN achieved higher success rates on most of the schemes than CNN, but CNN was faster most of the time.

TABLE X. ATTACK RESULTS BY KNN AND CNN.

Schemes	Success rate		Speed(s)	
	KNN	CNN	KNN	CNN
reCAPTCHA	77.2%	38.4%	10.27	10.19
Yahoo!	5.0%	5.2%	28.56	23.81
Baidu	44.2%	46.6%	2.81	2.21
Wikipedia	23.8%	20.4%	3.74	2.90
QQ	56.0%	22.4%	4.95	4.61
Microsoft	16.2%	8.6%	12.59	6.64
Amazon	25.8%	20.2%	13.18	8.68
Taobao	23.4%	20.4%	4.64	5.25
Sina	9.4%	4.4%	4.83	5.21
Ebay	58.8%	32.6%	5.98	5.50

VII. Is there a Defence?

In principle, some countermeasures may circumvent our attack to an extent, by mitigating key steps of the attack.

Mitigating component extraction by overlapping, i.e. make adjacent characters overlap to prevent segmentation, or by rotating, i.e. rotate characters to some certain angles, making some strokes of adjacent characters connect or overlap.

Clear directional information is important for our directional filtering. If characters are connected or overlapped too much, the connected strokes will make it harder for our component extraction to work. Rotation can have a similar impact on our attack.

Mitigating partition and recognition by a variety of methods, such as increasing the length of Captcha or adopting a varied length, and using a large alphabet set. These methods will make the solution space larger, likely resulting in a decreased attack speed and success rate. Warping characters and introducing noise arcs will increase recognition difficulty.

We empirically evaluate some most promising countermeasures as follows.

We chose Baidu, Taobao and Amazon as the representative schemes respectively. For each experiment, 500 randomly collected Captchas were used as a sample set, and another 500 randomly chosen samples as a test set.

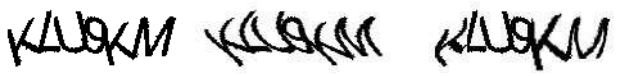
1) Overlapping. Overlapping removes space between characters and makes them overlapped, and it is considered by far the most secure anti-segmentation technique [5].

We use the Baidu scheme as a case study to evaluate the effectiveness of overlapping. We modify the original Captchas by increasing character overlapping by 1, 2 and 3 pixels, respectively, and then run our attack on them. Our new success rates are 21.2%, 15.2% and 8.4%, respectively, while the original is 44.2%. The more overlapped the characters, the less successful our attack became.

2) Rotating. To evaluate the effectiveness of rotating as a defence, we chose the Taobao scheme for an experiment. We rotated one or more characters to make adjacent strokes of different characters connected or overlapping, but kept the Captcha length and characters unchanged. The success rate our attack achieved on the hardened test set is 7.8the original set. This indicates rotating does have a positive effect in enhancing security.

3) Warping. Warping has two forms: global warping that transforms the whole Captcha string globally, and local warping that acts on some of the characters.

We tested both forms of warping on the Amazon scheme (see Figure 9). With global and local warping applied, respectively, the success rate of our attack dropped from the original 25.8% to 5.4% and 8.8%, respectively.



(a) Original (b) Global warping (c) Local warping

Fig. 9. Warping defense on Amazon Captcha.

4) Combining countermeasures. We also perform a new set of experiments to test various combinations of the countermeasures, and evaluate each combination's resistance to our attack, in the aim of making a further insight into the strength of combining these mechanisms. There are four different combinations of these countermeasures and we test all of them. To achieve consistent and rigorous results, our experiments test all the countermeasure combinations on a single scheme. We choose Amazon for our experiments, and the size of both our sample set and test set is 500.

Table XI summarises our experiment results, listing each countermeasure and combination along with its influence on the Captcha's resistance to our attack.

The results clearly suggest the following. First, the combination of two countermeasures is indeed more secure than each single countermeasure alone. The combination of three countermeasures achieves the best defence. Among all single countermeasure, warping is the most secure one. However, warping is a double-edged sword; it indeed enhances security, but too much warping will significantly decrease usability. What a level of warping is good to strike the right balance between security and usability is an important issue for Captcha designers to consider.

Although all the above countermeasures achieve a reduction of our success rates, our attack still has broken all the hardened schemes, since it has achieved a success rate of higher than 1% for each of them. Therefore, these mechanisms are at most partial defences. On the other hand, in performing our empirical studies, we did not consider and evaluate the impact of these defence mechanisms on usability. However, it is important to strike the right balance between security and usability in Captcha design [26]. It remains an open problem what design will be simultaneously usable and robust to our attack.

VIII. Summary and Conclusion

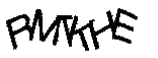
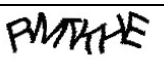
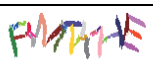
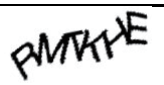
We have proposed a simple attack on text Captchas. Tested on real-world Captchas deployed by top 20 most popular websites, and on several Captchas that were generally considered hard, our attack has broken them all, mostly with a good success. Although our success rates on a few schemes are relatively low, we believe that our attack's general applicability trumps very high performance. It is more important to be able to break any novel scheme to some extent, than to break a single scheme very well.

If an attacker aims to break a particular scheme, ad hoc attacks might indeed achieve a better success rate than our generic attack. But when the attacker aims to break multiple schemes, our generic attack means a much better cost-effectiveness.

In contrast to the common practice of Captcha robustness analysis, which is based on a toolbox approach, our attack uses a single segmentation method, and a single recognition strategy. Our attack is simple, fast and generic, and because of these characteristics, it is probably the best attack so far.

Our attack is based on a novel application of 2D Log-Gabor filters. The key insight and innovation that differentiates our attack from prior art is the following. No matter Captcha texts

TABLE XI. COUNTERMEASURE COMBINATIONS.

Experiments	Sample Image	Reconstruction Image	Overlapping	Rotating	Warping	Attack Success
1			✓			11.6%
2				✓		13%
3					✓	8.8%
4			✓	✓		7.6%
5			✓		✓	7.4%
6				✓	✓	6.8%
7			✓	✓	✓	1.4%

are connected or not, and no matter they use hollow fonts or not, Log Gabor provides a uniform and effective method for breaking the images into a small number of meaningful pieces, i.e. character strokes, in a structured way. These pieces then can be assembled to reconstruct correct characters with an intelligent algorithm.

It is known for long that simple cells in the visual cortex of mammalian brains can be modeled by Gabor functions [10, 11]. That is to say, perception in the human visual system is more or less similar to image analysis with Gabor filters. These profound insights help to explain the power of our attack, and the failure of common text Captchas that we have analysed: our humans' Captcha-solving process can be computationally approximated by our Gabor filter based recognition approach. When computers can reliably approximate via an automated algorithm humans' solving process, certainly such Captcha designs are doomed. However, to reach this simple observation, it takes many years of hard work.

Since the invention of Captcha technology in early 2000, an open problem that is important for security has been outstanding in the research communities and intrigued researchers for 15 years: is there an effective but general attack that breaks all (representative) text schemes? The implication of resolving this open problem is apparent: are we on the wrong direction in text Captcha design? Characters are distorted harder everyday, but is this really necessary, or just making a legitimate user's life harder? Our attack is a step forward towards resolving this long-standing problem, and contributes to debates around its implications.

A full defence against our attack is an interesting but challenging open problem, which we share with the whole community. We expect our work to inspire novel attacks and defences, as well as innovative designs in this interesting interdisciplinary area.

Given the practical relevance and intellectual interest of the Captcha technology, it is important to ask: Are text Captchas dead? Our answer is both yes and no. On the one hand, as illustrated by our attack, the common practice of text Captcha designs is certainly dubious and shaking. On the other hand, we believe innovations will be able to bring out next generation of text Captchas that are more usable and more secure than its predecessors. We encourage both the research community and the industry to ponder and discuss: what is the next step for text Captchas? Or, is it now the time to take alternative solutions such as image recognition Captchas more seriously?

ACKNOWLEDGEMENTS

We thank Ross Anderson, John Daugman, Jussi Palomäki and Will Ng for helpful conversations, and thank Venkat Venkatakrishnan and anonymous reviewers for constructive comments. Xidian authors are supported by the National Natural Science Foundation of China (61472311) and the Fundamental Research Funds for the Central Universities.

REFERENCES

[1] A S El Ahmad, J Yan, and L Marshall. The robustness of a new captcha. In *Proceedings of the Third European Workshop on System Security*, pages 36–41. ACM, 2010.

[2] A S El Ahmad, J Yan, and M Tayara. *The robustness of Google CAPTCHAs*. Computing Science, Newcastle University, 2011.

[3] Alexa. Alexa top 500 global sites. https://www.alexa.com/topsites.

[4] E Bursztein, J Aigrain, A Moscicki, and J C Mitchell. The end is nigh: generic solving of text-based captchas. In *8th USENIX Workshop on Offensive Technologies(WOOT 14)*, San Diego, CA, August 2014. USENIX Association.

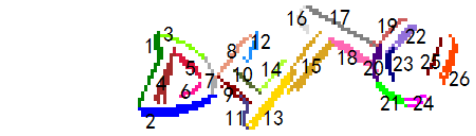
[5] E Bursztein, M Martin, and J Mitchell. Text-based captcha strengths and weaknesses. In *CCS'11*, pages 125–138. ACM, 2011.

[6] E Bursztein, A Moscicki, C Fabry, S Bethard, J C Mitchell, and D Jurafsky. Easy does it: more usable captchas. In *CHI'14*, pages 2637–2646. ACM, 2014.

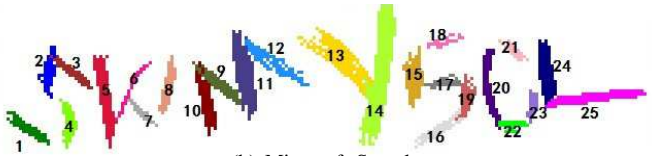
[7] K Chellapilla, K Larson, P Y Simard, and M Czerwinski. Building segmentation based human-friendly human interaction proofs, 2005.
[8] Z Dang, J Lei, and J Lan. A method of constructive captcha based on gabor sub-space. *Journal of Computational Information Systems*, 9(8):3093–3099, 2013.
[9] J Daugman. Uncertainty relation for resolution in space, spatial frequency, and orientation optimized by two-dimensional visual cortical filters. *JOSA A*, 2(7):1160–1169, 1985.
[10] J Daugman. Probing the uniqueness and randomness of iriscodes: Results from 200 billion iris pair comparisons. *Proceedings of the IEEE*, 94(11):1927–1935, 2006.
[11] D J Field. Relations between the statistics of natural images and the response properties of cortical cells. *JOSA A*, 4(12):2379–2394, 1987.
[12] W T Freeman and E H Adelson. The design and use of steerable filters. *IEEE Transactions on PAMI*, 13(9):891–906, 1991.
[13] H Gao, W Wang, J Qi, X Wang, X Liu, and J Yan. The robustness of hollow captchas. In *CCS'13*, pages 1075–1086. ACM, 2013.
[14] P Golle. Machine learning attacks against the asirra captcha. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 535–542. ACM, 2008.
[15] I J Goodfellow, Y Bulatov, J Ibarz, S Arnoud, and V Shet. Multi-digit number recognition from street view imagery using deep convolutional neural networks. *arXiv preprint arXiv:1312.6082*, 2013.
[16] Y Lecun. The mnist database of handwritten digits algorithm results. http://yann.lecun.com/exdb/mnist/.
[17] M Mohamed, N Sachdeva, M Georgescu, S Gao, N Saxena, C Zhang, P Kumaraguru, P C van Oorschot, and W B Chen. A three-way investigation of a game-captcha: automated attacks, relay attacks and usability. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 195–206. ACM, 2014.
[18] G Mori and J Malik. Recognizing objects in adversarial clutter: Breaking a visual captcha. In *CVPR'03*, volume 1, pages I–134. IEEE, 2003.
[19] P Y Simard. Using machine learning to break visual human interaction proofs (hips). In *NIPS'04*, 2004.
[20] J Tam, J Simsa, S Hyde, and L V Ahn. Breaking audio captchas. In *Advances in Neural Information Processing Systems*, pages 1625–1632, 2008.
[21] K Thomas, D McCoy, C Grier, A Kolcz, and V Paxson. Trafficking fraudulent accounts: the role of the underground market in twitter spam and abuse. In *USENIX Security Symposium*, 2013.
[22] Vicarious. Vicaricous. http://vimeo.com/77431982.
[23] Y Xu, G Reynaga, S Chiasson, J M Frahm, F Monrose, and P C van Oorschot. Security and usability challenges of moving-object captchas: Decoding codewords in motion. In *USENIX Security Symposium*, pages 49–64, 2012.
[24] J Yan and A S El Ahmad. Breaking visual captchas with naive pattern recognition algorithms. In *ACSAC'07*, pages 279–291. IEEE, 2007.
[25] J Yan and A S El Ahmad. A low-cost attack on a microsoft captcha. In *CCS'08*, pages 543–554. ACM, 2008.
[26] J Yan and A S El Ahmad. Usability of captchas or usability issues in captcha design. In *SOUPS'08*, pages 44–52. ACM, 2008.
[27] B B Zhu, J Yan, Q Li, C Yang, J Liu, N Xu, M Yi, and K Cai. Attacks and design of image recognition captchas. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 187–200. ACM, 2010.

APPENDIX

Here we present the details of key steps like graph building, pruning and searching for the QQ and Microsoft schemes. Figure 10 shows QQ and Microsoft challenges with all components rank ordered.



(a) QQ Sample



(b) Microsoft Sample

Fig. 10. All components rank ordered.

TABLE XII. THE INITIAL $n \times n$ TABLE FOR QQ CAPTCHA.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1		•	•	•	•	•	•							
2		•	•	•	•	•	•							
3														
4								•	•					
5								•	•	•	•	•		
6										•	•	•		
7													•	•
8...12														

	15	16	17	18	19	20	21	22	23	24	25	26
13	•	•										
14	•	•										
15			•	•								
16			•	•								
17												
18					•	•	•	•	•			
19												•
20												•
21												•
22...26												

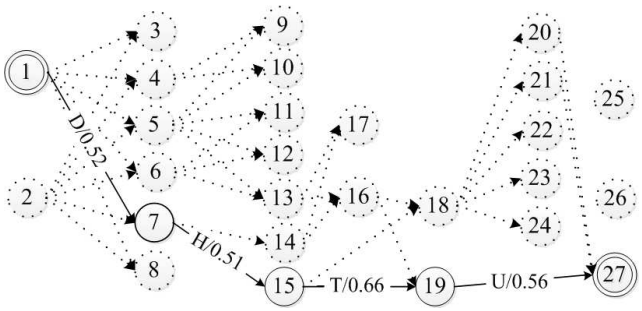


Fig. 11. The search graph for QQ sample.

Tables XII show the corresponding initial tables for the QQ sample, and Figure 11 is its search graph. Only one path survives pruning for the QQ sample, which simplifies the search process to an extreme.

TABLE XIII. THE INITIAL $n \times n$ TABLE FOR MICROSOFT CAPTCHA.

	1	2	3	4	5	6	7	8	9	10	11	12	
1		•	•	•	•								
2			•	•	•	•							
3					•	•	•						
4					•	•	•	•					
5						•	•	•					
6								•	•				
7								•	•	•			
8									•	•	•		
9										•	•	•	
10										•	•	•	
11												•	
12												•	

	13	14	15	16	17	18	19	20	21	22	23	24	25
13	•	•											
14			•	•	•	•	•						
15				•	•	•	•	•					
16				•	•	•	•	•	•	•			
17							•	•	•	•	•		
18							•	•	•	•	•		
19									•	•	•	•	
20											•	•	
21												•	
22												•	
23													
24													•
25													•

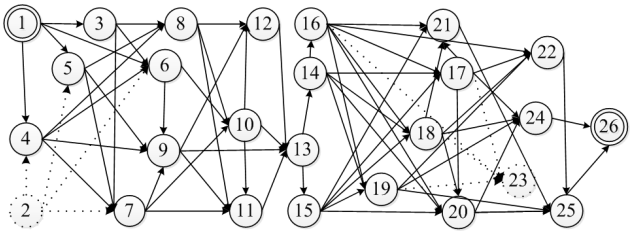


Fig. 12. The search graph for Microsoft sample.

Tables XIII and XIV show the corresponding initial and final tables for the Microsoft sample. Figure 12 describes the search graph and Table XV gives the dynamic search process for the Microsoft sample.

TABLE XIV. THE FINAL $n \times n$ TABLE FOR MICROSOFT CAPTCHA.

	1	2	3	4	5	6	7	8	9	10	11	12	
1		Q/0.36	5/0.49	***5/0.7***	3/0.44								
2													
3					T/0.42	y/0.4	y/0.34						
4					H/0.58	W/0.49	W/0.36	P/0.31					
5						***V/0.84***	V/0.6	L/0.51					
6								H/0.44	N/0.11				
7								***J/0.49***	N/0.46	X/0.38			
8									H/0.49	L/0.44	N/0.46		
9										V/0.25	H/0.22	***M/0.92***	
10										4/0.2	N/0.5	M/0.47	
11												T/0.52	
12												X/0.2	
	13	14	15	16	17	18	19	20	21	22	23	24	25
13	4/0.33	***y/0.87***											
14			T/0.21	W/0.48	W/0.46	W/0.44	L/0.44						
15				L/0.46	L/0.39	5/0.43	***5/0.91***	V/0.36					
16				y/0.46	y/0.38	5/0.36	5/0.54	Y/0.31	X/0.32				
17							y/0.39	y/0.43	L/0.31		D/0.27		
18							Y/0.38	y/0.37	L/0.27		D/0.37		
19									F/0.4		C/0.34	P/0.38	
20											***C/0.75***	H/0.38	
21												y/0.37	
22												y/0.43	
23													
24													***L/0.86***
25													D/0.56

TABLE XV. THE SEARCH PROCESS FOR MICROSOFT CAPTCHA.

j	$step[j]$	$value[j]$	$result[j]$	j	$step[j]$	$value[j]$	$result[j]$	j	$step[j]$	$value[j]$	$result[j]$
3	1	0.36	Q	16	5	2.68	5LM4T	21	5	3.37	5LMyV
4	1	0.49	5		6	3.5	5VJM4T		6	4.19	5VJMyV
5	1	0.7	5		7	3.35	5VJVT4T		7	4.73	5VJMyLy
6	1	0.44	3	17	5	3.48	5LMyL		8	4.59	5VJVTyLy
	2	1.08	5H		6	4.3	5VJMyL		9	4.25	5VJVT4Tyy
7	2	1.55	5V		7	4.15	5VJVTyL	22	6	3.85	5LMy5F
8	2	1.31	5V		8	3.82	5VJVT4Ty		7	4.67	5VJMy5F
9	2	1.22	5L	18	5	3.41	5LMyL		8	4.52	5VJVTy5F
	3	2.04	5VJ		6	4.23	5VJMyL		9	4.14	5VJVT4TyL
10	2	0.55	3N		7	4.08	5VJVTyL	24	6	4.68	5LMy5C
	3	2.01	5VN		8	3.74	5VJVT4Ty		7	5.5	5VJMy5C
11	3	1.93	5VX	19	5	3.44	5LMy5		8	5.46	5VJMyLyC
	4	2.29	5VJV		6	4.27	5VJMy5		9	5.31	5VJVTyLyC
12	3	1.78	5VN		7	4.12	5VJVTy5		10	4.97	5VJVT4TyyC
	4	2.52	5VNN		8	3.72	5VJVT4T5	25	6	4.3	5LMy5H
13	3	2.14	5LM	20	5	3.92	5LMy5		7	5.12	5VJMy5H
	4	2.96	5VJM		6	4.74	5VJMy5		8	5.1	5VJMyLyy
	5	2.81	5VJVT		7	4.7	5VJMyLy		9	4.96	5VJVTyLyy
14	4	2.47	5LM4		8	4.55	5VJVTyLy		10	4.62	5VJVT4Tyyy
	5	3.29	5VJM4		9	4.21	5VJVT4Tyy	26	7	5.54	5LMy5CL
	6	3.14	5VJVT4						***8***	***6.37***	***5VJMy5CL***
15	4	3.01	5LMy						9	6.32	5VJMyLyCL
	5	3.83	5VJMy						10	6.18	5VJVTyLyCL
	6	3.68	5VJVTy								