Seeing is Not Believing: Camouflage Attacks on Image Scaling Algorithms

Qixue Xiao* [1,4], Yufei Chen* [2,4], Chao Shen[2], Yu Chen[1,5], and Kang Li[3]
[1]*Department of Computer Science and Technology, Tsinghua University*
[2]*School of Electronic and Information Engineering, Xi'an Jiaotong University*
[3]*Department of Computer Science, University of Georgia*
[4]*360 Security Research Labs*
[5]*Peng Cheng Laboratory*

Abstract

Image scaling algorithms are intended to preserve the visual features before and after scaling, which is commonly used in numerous visual and image processing applications. In this paper, we demonstrate an automated attack against common scaling algorithms, i.e. to automatically generate camouflage images whose visual semantics change dramatically after scaling. To illustrate the threats from such camouflage attacks, we choose several computer vision applications as targeted victims, including multiple image classification applications based on popular deep learning frameworks, as well as mainstream web browsers. Our experimental results show that such attacks can cause different visual results after scaling and thus create evasion or data poisoning effect to these victim applications. We also present an algorithm that can successfully enable attacks against famous cloud-based image services (such as those from Microsoft Azure, Aliyun, Baidu, and Tencent) and cause obvious misclassification effects, even when the details of image processing (such as the exact scaling algorithm and scale dimension parameters) are hidden in the cloud. To defend against such attacks, this paper suggests a few potential countermeasures from attack prevention to detection.

1 Introduction

Image scaling refers to the resizing action on a digital image, while preserving its visual features. When scaling an image, the downscaling (or upscaling) process generates a new image with a smaller (or larger) number of pixels compared to the original one. Image scaling algorithms are widely adopted in various applications. For example, most deep learning computer vision applications use pre-trained convolutional neural network (CNN) models, which take data with a fixed size defined by the input layers of those models. Hence, input images have to get scaled in the data preprocessing procedure to meet with specific model input size. Popular deep learning frameworks, such as Caffe [17], TensorFlow [13] and Torch [26], all integrate various image scaling functions in their data preprocessing modules. The purpose of these built-in scaling functions is to allow the developers to use these frameworks to handle images that do not match the model's input size.

Although scaling algorithms are widely used and are effective to normal inputs, the design of common scaling algorithms does not consider malicious inputs that may intentionally cause different visual results after scaling and thus change the "semantic" meaning of images. In this paper, we will see that an attacker can utilize the "data under-sampling" phenomena occurring when a large image is resized to a smaller one, to cause "visual cognition contradiction" between human and machines for the same image. In this way, the attacker can achieve malicious goals like detection evasion and data poisoning. What's worse, unlike adversarial examples, this attack is independent from machine learning models. The attack indeed happens before models consume inputs, and hence this type of attacks affects a wide range of applications with various machine learning models.

This paper characterizes this security risk and presents a camouflage attack on image scaling algorithms (abbreviated as *scaling attack* in the rest of the paper). To successfully launch a scaling attack, attackers need to deal with two technical challenges: (a) First, an adversary needs to decide where to insert pixels with deceiving effects by analyzing the scaling algorithms. It is tedious and practically impossible to use manual efforts to determine exact pixel values to achieve a desired deceiving effect for realistic images. Therefore, a successful attack needs to explore an automatic and efficient camouflage image generation approach. (b) Second, for cloud-based computer vision services, the exact scaling algorithm and input size of their models are transparent to users. Attackers need to infer scaling-related parameters of the underneath algorithms in order to successfully launch such attacks.

To overcome these challenges, we first formalize the process of scaling attacks as a general optimization problem. Based on the generalized model, we propose an automatic

*Co-first authors. This work was completed during their internship program at 360 Security Research Labs.

generation approach that can craft camouflage images efficiently. Moreover, this work examines the feasibility of this attack in both the white-box and black-box scenarios, including applications based on open deep learning frameworks and commercial cloud services:

- In the white-box case (see Section 6.1 for more details), we analyze common scaling implementations in three popular deep learning frameworks: Caffe, TensorFlow and Torch. We find that nearly all default data scaling algorithms used by these frameworks are vulnerable to the scaling attack. With the presented attack, attackers can inject poisoning or deceiving pixels into input data, which are visible to users but get discarded by scaling functions, and eventually being omitted by deep learning systems.
- In the black-box case (see Section 6.2 for more details), we investigate the scaling attack against cloud-based computer vision services. Our results show that even when the whole image processing pipeline and design details are hidden from users, it is still possible to launch the scaling attack to most existing cloud-based computer vision services. Since image scaling modules are built upon open image processing libraries or open interpolation algorithms, possible ways of image scaling implementation are relatively limited. Attackers can design a brute-force testing strategy to infer the scaling algorithm and the target scale. In this paper, we exhibit a simple but efficient testing approach, with successful attack results on Microsoft Azure[1], Baidu[2], Aliyun[3] and Tencent[4].
- Interestingly, we also discover and discuss the range of the attacking influence extends to some computer-graphic applications, such as mainstream web browsers shown in Section 6.3.

We provide a video to demonstrate the attack effects, which is available at the following URL: https://youtu.be/Vm2N0mb14Ow.

This paper studies the commonly used scaling implementations, especially for image scaling algorithms employed in popular deep learning frameworks, and reveals potential threats to the image scaling process. Our contributions can be summarized as follows:

- This paper reveals a security risk in image scaling process in computer vision applications. We validate and testify the image scaling algorithms commonly used in popular deep learning (DL) frameworks, and our results indicate that the security risk affects almost all image applications based on DL frameworks.
- This paper formalizes the scaling attack into a constrained optimization problem, and presents the corresponding implementation to generate camouflage images automatically and efficiently.
- Moreover, we prove that the presented attack is still effective for cloud vision services, where the implementation details of image scaling algorithms and parameters are hidden from users.
- To eliminate the threats from the scaling attack, we suggest several potential defense strategies from two aspects: attack prevention and detection.

[1] https://azure.microsoft.com/en-us/services/cognitive-services/computer-vision/?v=18.05

[2] https://ai.baidu.com/tech/imagerecognition/fine_grained

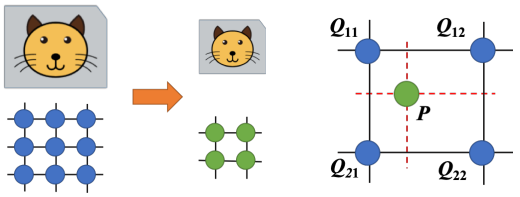
[3] https://data.aliyun.com/ai?spm=a2c0j.9189909.810797.11.4aae547aEqltqh#/image-tag

[4] https://ai.qq.com/product/visionimgidy.shtml#tag

2 Image Scaling Attack Concept and Attack Examples

In this section, we first present a high level overview of image scaling algorithms, followed by the concept of image scaling attacks. Then, we exhibit some examples of the image scaling attack, and finally we conduct an empirical study of the image scaling practices in deep learning based image applications.

2.1 An Overview of Image Scaling Algorithms



(a) Image scaling. (b) Interpolation in scaling.

Figure 1: The concept of image scaling.

Image scaling algorithms are designed to preserve the visual features of an image while adjusting its size. Fig.1 presents the general concept of a common image scaling process. A scaling algorithm infers value of each "missing point" by using interpolation methods. Fig.1b shows an example of constructing pixel P in the output image based on the pixels of Q_{11}, Q_{12}, Q_{21} and Q_{22} in the original image. A scaling algorithm defines which neighbor pixels to use in order to construct a pixel of the output image, determines the relative weight values assigned to each individual neighbor pixels. For example, for each pixel in the output image, a nearest neighbor algorithm only picks a single pixel (the nearest one) from the input to replace it. A bilinear algorithm considers a set of neighbor pixels surrounding the target pixel as the

Figure 2: An example showing deceiving effect of the scaling attack. (Left-side: what humans see; right-side: what DL models see)

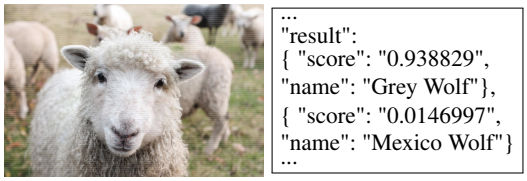


Figure 3: A scaling attack example against Baidu image classification service.

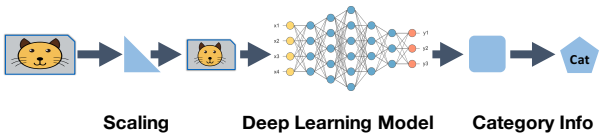


Figure 4: How data get processed in image classification systems.

basis. It then calculates a weighted average of the neighbor pixel values as the value assigned to the target pixel.

Such scaling algorithms often assume that pixel values in an image are results from natural settings, and they do not anticipate pixel-level manipulations with malicious intents. This paper demonstrates that an attacker can use scaling algorithms to alter an image's semantic meaning by carefully adjusting pixel-level information.

2.2 Attack Examples

The scaling attack potentially affects all applications that apply scaling algorithms to preprocess input images. To demonstrate potential risks and deceiving effects of the scaling attack, here we provide two attack examples of the scaling attack on practical applications.

Fig.2 presents the first attack example for a local image classification application *cppclassification* [16], a sample program released by the Caffe framework. For the classification model with an input size of 224*224, we specially craft input images of a different size (672*224). The image in the left column of Fig.2 is one input to the deep learning application, while the image in the right column is the output of the scaling function, i.e., the *effective image* fed into the deep learning model. While the input in the left column of Fig.2 visually presents a sheep-like figure, the deep learning model takes the image in the right column as the actual input and classifies it as an instance of "White Wolf".

To validate the deceiving effect on deep learning applications, we build one image classification demo based on the BAIR/BVLC GoogleNet model [8], which assumes the input data are of the scale of 224*224. When an image with a different size is provided, the application triggers the native resize() function built in the Caffe framework to rescale the image to fit the input size of the model (224*224). The exact classification setup details and the program outputs are presented in Appendix A.

Fig.3 exhibits one attack example against the Baidu image classification service. The attack image is crafted from a sheep image, with the aim to lead people to regard it as a sheep but the machine to regard it as a wolf. The results returned by the cloud service API[5] show that the attack image is classified as the "Grey Wolf" with a high confidence score (achieving 0.938829), indicating that our attack is effective. More examples of the scaling attack against more cloud-based computer vision services are presented in Table 3. In fact, image scaling algorithms are commonly used by a wide range of computer-graphic applications, rather than limited to deep-learning-based computer vision systems. Therefore, they are all potentially threatened by this type of security risk.

2.3 Empirical Study of Image Scaling Practices in Deep Learning Applications

Data scaling is actually a common action in deep learning applications. Fig.4 shows how the scaling process is involved in open-input applications' data processing pipelines, such as image classification as an Internet service. For design simplicity and manageable training process, a deep learning neural network model usually requires a fixed input scale. For image classification models, images are usually resizedd to 300*300 to ensure high-speed training and classification. As shown in Table 1, we examine nine popular deep learning models and all of them use a fixed input scale for their training and classification process.

For deep learning applications that receive input data from fixed input sources, such as video cameras, the input data formats are naturally determined. Even in such situation, we find that the image resizing is still needed in certain cases.

One common situation we observe is the use of pre-trained models. For example, NVIDIA offers multiple self-driving sample models [6], and all these models use a specific input

[5]The original API response is presented in Chinese. Here we translate it into English.

Table 1: Input sizes of various deep learning models.

Model	Size (pixels*pixels)
LeNet-5	32*32
VGG16, VGG19, ResNet, GoogleNet	224*224
AlexNet	227*227
Inception V3/V4	299*299
DAVE-2 Self-Driving	200*66

size 200*66. However, for the recommended camera [24] specification provided by NVIDIA, the size of generated images varies from 320*240 to 1928*1208. None of the recommended cameras produce output that matches the NVIDIA's model input size. Therefore, for system developers that do not want to redesign or retrain their models, they have to employ scaling functions in their data processing pipeline to fit the pre-trained model's input scale. Recent research work, such as the sample applications used in DeepXplore [25], also shows that the resizing operation is commonly used in self-driving applications to adjust original video frames' size to the input size of models.

Most deep learning frameworks provide data scaling functions, as shown in Table 2. Programmers can handle images with different sizes without calling scaling function explicitly. We examined several sample programs released by popular deep learning frameworks, such as Tensorflow, Caffe, and Torch, and we have found all of them implicitly trigger scaling functions in their data processing pipelines. Appendix B provides several examples.

3 Related Work

This section briefly reviews some related work and makes a comparison with our approach.

3.1 Information Hiding

Information hiding is a significant topic in information security [2,9,15,18,19,21,27,30,31]. Information hiding methods achieve reversible data hiding by image interpolation, but these are different from our attack method: First, the goals are different. The information hiding methods conceal data in a source image to make the secret information unnoticed by human, and image applications operate on the complete data. Our presented attack hides a target image in a source image to cause a visual cognition contradiction between human and image applications. The core components (such as deep learning classifiers) of the victim applications operate on a partial data (i.e. the scaling output). Second, information hiding efforts often impose a customized coding method (such as LSB and NIP [21]) in order to conceal and recover hidden information. This coding scheme is often kept as a secret known only to the designer of the specific information hiding method. In contrast, a scaling attack is based on the interpolation algorithm built within the victim application to achieve the deceiving effect. The main task for an attacker is to reverse engineering the scaling algorithms and design the pixel replacement policy.

3.2 Adversarial Examples

The research of adversarial examples attract growing public attentions with the reviving popularity of Artificial Intelligence. An adversarial image fools the Artificial Intelligence by inserting perturbations into the input image, which are hard to be noticed by human eyes. For example, Goodfellow *et al.* [14] presented a linear explanation of adversarial examples and revealed that such attack is effective for current sufficiently linear deep networks. In addition to the theoretical analysis, Alexey *et al.* [20] added the perturbations into the physical world and successfully launched the attack. It should be noted that the attack target of existing adversarial examples essentially aims at machine learning models, while our method focuses on the data preprocessing step, concretely, the image scaling action. Vulnerabilities in code implementation, such as control-flow hijacking, also could lead to recognition evasions [32]. However, we exploit the weakness of scaling algorithms in this work other than code implementation.

3.3 Invisible/Inaudible Attacks

Some researchers investigate potential attacks beyond the human sensing ability. Ian Markwood *et al.* [22] showed a content masking attack against the Adobe PDF standard. By tampering the font files, the attacker can insert secret information into PDF files without being noticed by human. They demonstrated such attack against state-of-the-art information-based services. Besides the attacks in vision fields, Zhang *et al.* [34] presented the *DolphinAttack* against speech recognition (SR) systems. They created secret voice commands on ultrasonic carriers that are inaudible for human beings, but can be captured and sensed by voice controllable systems. In this way, an attacker can control SR systems "silently". It has been proved that the widely used SR systems, like Apple Siri and Google Now, are vulnerable to such attack. Our attack is like a reverse of such invisible/inaudible attacks. The attacker leverage the difference between the input and output of the scaling function. Most part of the content visible to human is not consumed by the component that uses the scaling output.

4 Formalizing the Scaling Attack

This section describes the goal of a typical scaling attack and how we design a method to automatically create attack images with deceiving effects. The autonomous attack image crafting

Table 2: Scaling algorithms in deep learning frameworks.

DL Framework	Library	Interpolation Method	Order [a]	Validation [b]
Caffe	OpenCV [7]	Nearest	H→V	✓
		Bilinear(☆)	H→V	✓
		Bicubic	H→V	✓
		Area	H→V	†
		Lanczos	H→V	†
Tensorflow [c]	Python-OpenCV Pillow [10] Tensorflow.image	Nearest(☆Pillow)	H→V	✓
		Bilinear (☆Python-OpenCV, Tensorflow.image)	H→V	✓
		Bicubic	H→V	✓
		Area	H→V	†
		Lanczos	H→V	†
Torch	Torch-OpenCV	Nearest	H→V	✓
		Bilinear(☆)	H→V	✓
		Bicubic	H→V	✓
		Area	H→V	†
		Lanczos	H→V	†

☆ The default scaling algorithm.

[a] H→V means the algorithm first scales horizontally and then vertically.

[b] The validation is performed on attack with a constraint $\varepsilon = 0.01$. ✓ represents generate attack images successfully satisfying the constraints. † means we have not yet verified the attack effects because the algorithm is complex and rarely used in DL applications. More details please see Section 6.1.

[c] Tensorflow integrates multiple image processing packages, including Python-OpenCV, Pillow, and Tensorflow.image.

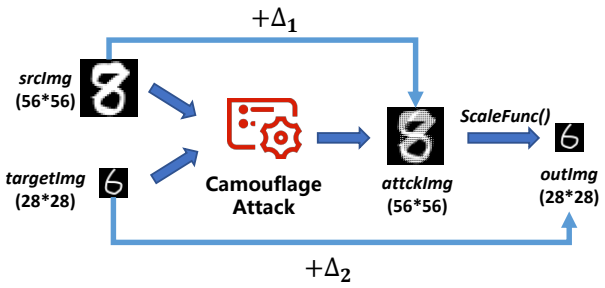


Figure 5: Automatic attack image crafting.

framework is shown in Fig.5, and details are presented in Section 4.2.

4.1 Attack Goals

The goal of the scaling attack is to create a deceiving effect with images. Here the deceiving effect refers to the case that an image partially or completely presents a different meaning to humans before and after a scaling action. In such case, we call the input file to the scaling action the *attack image*.

To describe the process of a scaling attack, we define the following four conceptual objects involved in one attack.

- source image (or *srcImg*) S_{m*n} – the image that an attacker wants the attack image to look like.
- attack image (or *attackImg*) A_{m*n} – the crafted image eventually created and fed to the scaling function.
- output image (or *outImg*) $D_{m'*n'}$ – the output image of the scaling function.
- target image (or *targetImg*) $T_{m'*n'}$ – the image that the attacker wants the *outImg* to look like.

In some cases, some of these objects are identical. For example, it is often possible for an attacker to generate an out image that is identical to the target image.

The process of performing a scaling attack is to craft an *attackImg* under visual similarity constraints with *srcImg* and *targetImg*. Based on the intent and constraints on source images, we define the image scaling attack into two attack modes.

The first attack mode is when both the source and target images are specified, i.e. the attacker wants to scale an image that looks like a specific source image to an image that looks like a specific target image. In this mode the attacker launches a **source-to-target attack**, where the semantics of *srcImg* and *targetImg* are controlled as he/she wants. However, posing constraints on the looks of both the source and target images makes this attack mode more challenging. We call this mode of attack the *strong attack form*.

The second attack mode is when only the target image is specified. In that case, the attacker just wants to cause a vision contradiction during image scaling, as long as it is related to a certain concept (such as any images of sheep). In some extreme cases, the image content could be meaningless, e.g., just to create a negative result to an image classifier. Without a specific source image, the attacker's goal is to increase the dissimilarity before and after image scaling as much as possible. In this mode, the similarity constraints get relaxed and we call this mode of attack the *weak attack form*.

4.2 An Autonomous Approach on Attack Image Generation

We are interested to develop a method to automatically create scaling attack images in both the strong and weak attack forms. In order to achieve such goal, we first formalize the description of the data transition process among the four conceptual objects, and then we seek an algorithmic solution to create attack images.

The relationship between the four conceptual objects can be described in the following formulas.

First, the transition between *srcImg* and *attackImg* can be represented by a perturbation matrix Δ_1, and so does the difference between *outImg* and *targetImg*. These transition can be represented by

$$A_{m*n} = S_{m*n} + \Delta_1 \tag{1}$$

For the transition between *attackImg* and *outImg*, we consider the scaling effect as a function ScaleFunc(), which converts an $m*n$ input image A_{m*n} to an $m'*n'$ output image $D_{m'*n'}$ [6].

$$\text{ScaleFunc}(A_{m*n}) = D_{m'*n'} \tag{2}$$

ScaleFunc() is a surjective function, i.e. there exist multiple possible inputs A_{m*n} that all result in the same output $D_{m'*n'}$.

To perform a scaling attack, attackers need to craft an attack image A_{m*n}, which is the source image S_{m*n} plus a perturbation matrix Δ_1. In the meanwhile, the scaling result of the attack image, i.e., the output image $D_{m'*n'}$, needs to be visually similar with the target image $T_{m'*n'}$. Here we use Δ_2 to evaluate the difference between $D_{m'*n'}$ and $T_{m'*n'}$.

$$\begin{aligned} &A_{m*n} = S_{m*n} + \Delta_1 \\ &\text{ScaleFunc}(A_{m*n}) = D_{m'*n'} \\ &D_{m'*n'} = T_{m'*n'} + \Delta_2 \end{aligned} \tag{3}$$

Let us consider the strong attack form of scaling attack as in Fig.5, where both source S_{m*n} and target image $T_{m'*n'}$ are specified. The attacker's task is to find an attack image A_{m*n} being able to cause deceiving effect. Considering Eq.3, we can find multiple possible candidate matrices as solutions for A_{m*n} that satisfy the whole set of formulas. This is due to the surjection effect of ScaleFunc(). What the attacker wants to find is the matrix that produces the best deceiving effect among all possible solutions for A_{m*n}. One strategy is to find an A that is the most similar with S, while limiting the difference between D and T within an upper bound.

To find the best deceiving effect, we theoretically define an objective function that compares all solutions of A_{m*n}. To seek an algorithmic solution, we choose the L-norm distance[7] to capture the pixel-level differences as an approximation for measuring how close two images are.

In the strong attack form, we want to minimize the difference between A_{m*n} and S_{m*n}, and limit the difference between $D_{m'*n'}$ and $T_{m'*n'}$ within a threshold. Consequently, when the source image S_{m*n} and target image $T_{m'*n'}$ are given, the best result can be found by solving the following objective function.

$$\begin{aligned} &A_{m*n} = S_{m*n} + \Delta_1 \\ &\text{ScaleFunc}(A_{m*n}) = D_{m'*n'} \\ &D_{m'*n'} = T_{m'*n'} + \Delta_2 \\ &||\Delta_2||_\infty \leq \varepsilon * IN_{max} \\ &\text{Objective function} : \min(||\Delta_1||^2) \end{aligned} \tag{4}$$

where IN_{max} is the maximum pixel value in the current image format.

For the weak attack form, i.e. only the target image $T_{m'*n'}$ is given, what an attacker wants to optimize is to pick the attack image that visually has the largest difference from the target image. Thus, the best result should be found by solving the following objective function:

$$\begin{aligned} &R_{m*n} = \text{ScaleFunc}(T_{m'*n'}) \\ &A_{m*n} = R_{m*n} + \Delta_1 \\ &\text{ScaleFunc}(A_{m*n}) = D_{m'*n'} \\ &D_{m'*n'} = T_{m'*n'} + \Delta_2 \\ &||\Delta_2||_\infty \leq \varepsilon * IN_{max} \\ &\text{Objective function} : \max(||\Delta_1||^2) \end{aligned} \tag{5}$$

Notice that in the above constraints, we apply ScaleFunc() twice. The first call of ScaleFunc() is actually scaling the target image to the size of the attack image, i.e., upscaling an image from dimension $m'*n'$ back to $m*n$.

5 Creating Scaling Attack Images

After building up the formalized model of the scaling attack, in this section we investigate how to generate attack images automatically.

[6] Conventionally, we say a matrix of $m*n$ dimension has m rows and n columns, while an image of $m*n$ size consists of m columns and n rows. For convenience sake, in this paper we use a matrix X_{m*n} of $m*n$ dimension to refer to "an $m*n$ image".

[7] In this paper, $||\cdot||$ denotes the L^2-norm, while $||\cdot||_\infty$ denotes the L^∞-norm.

5.1 Empirical Analysis of Scaling Functions

In our implementation, we first need to find an appropriate expression of ScaleFunc(). We studied the implementation details of commonly used image processing packages. All of the scaling functions we studied perform the interpolation in two steps, one direction in each step. We design our attack algorithm with the assumption that the target scaling algorithm first resizes inputs horizontally and then vertically. Empirically the popular algorithms take this order (see Table 2, and more detailed analysis and examples are provided in Appendix C.). In case the scaling algorithm takes vertical order first, the attack method just needs to change accordingly. Hence, the ScaleFunc() in Eq.2 can be presented as:

$$\text{ScaleFunc}(X_{m*n}) = CL_{m'*m} * X_{m*n} * CR_{n*n'} \tag{6}$$

In Eq.6, $CL_{m'*m}$ and $CR_{n*n'}$ are two constant coefficient matrices determined by the interpolation algorithm, related to horizontal scaling ($m*n \rightarrow m*n'$) and vertical scaling ($m*n' \rightarrow m'*n'$), respectively.

With Eq.4 and Eq.6, we eventually build a relationship among the source image, the target image, and the perturbation:

$$\begin{aligned} &CL_{m'*m} * (S_{m*n} + \Delta_1) * CR_{n*n'} = D_{m'*n'} \\ &D_{m'*n'} = T_{m'*n'} + \Delta_2 \end{aligned} \tag{7}$$

5.2 Attack Image Crafting: An Overview

The main idea of automated scaling attack generation is to craft the attack image by two steps. The first step is to obtain the coefficient matrices in Eq.7. Section 5.3 presents a practical solution to find CL and CR, implemented as GetCoefficient(). The second step is to find the perturbation matrix Δ_1 to craft the attack image. We perform the attack image generation along each direction, in **reverse** order that we assume ScaleFunc() uses. Further, we decompose the solution of the perturbation matrix into the solution of a few perturbation vectors. By this way, we can significantly reduce the computational complexity for large size images. Section 5.4 provides more details of the second step, based on which we implement GetPerturbation() to find the perturbation vectors. Algorithm 1 and Algorithm 2 illustrate the attack image generation in the weak attack form and the strong attack form, respectively.

Weak attack form (Algorithm 1) [8]. First, we obtain the coefficient matrices by calling GetCoefficient() (line 2), which receives the size of S_{m*n} and $T_{m'*n'}$, and returns coefficient matrices $CL_{m'*m}$ and $CR_{n*n'}$, and then generate an intermediate source image $S^*_{m*n'}$ from $T_{m'*n'}$. Then, we call GetPerturbation(), which receives the column vectors from $S^*_{m*n'}$ and $T_{m'*n'}$, with the coefficient matrix CL and the object option ('max'), and returns the optimized perturbation matrix Δ^v_1, to solve the perturbation matrix column-wisely and craft out

[8] To clarify, here we use $X[i,:]$ and $X[:,j]$ to represent the i-th row and j-th column of matrix X respectively.

Algorithm 1 Image Crafting of the Weak Attack Form

Input: scaling function ScaleFunc(), target image $T_{m'*n'}$, source image size $(width_s, height_s)$, target image size $(width_t, height_t)$
Output: attack image A_{m*n}

1: $m = height_s$, $n = width_s$, $m' = height_t$, $n' = width_t$
2: $CL_{m'*m}$, $CR_{n*n'}$ = GetCoefficient(m, n, m', n')
3: $\Delta^v_1 = \mathbf{0}_{m*n'}$ ▷ Perturbation matrix of vertical attack.
4: $S^*_{m*n'}$ = ScaleFunc($T_{m'*n'}$) ▷ Intermediate source image.
5: **for** $col = 0$ to $n'-1$ **do**
6: $\Delta^v_1[:,col]$ = GetPerturbation($S^*[:,col]$, $T[:,col]$, CL, obj='max') ▷ Launch the vertical scaling attack.
7: **end for**
8: $A^*_{m*n'}$ = unsigned int($S^* + \Delta^v_1$)
9: S_{m*n} = ScaleFunc($T_{m'*n'}$) ▷ Final source image.
10: $\Delta^h_1 = \mathbf{0}_{m*n}$ ▷ Perturbation matrix of horizontal attack.
11: **for** $row = 0$ to $m-1$ **do**
12: $\Delta^h_1[row,:]$ = GetPerturbation($S[row,:]$, $A^*[row,:]$, CR, obj='max') ▷ Launch the horizontal scaling attack.
13: **end for**
14: A_{m*n} = unsigned int($S + \Delta^h_1$)
15: **return** A_{m*n} ▷ Get the crafted attack image.

$A^*_{m*n'}$ (line 5 to 8). Similarly, we solve another perturbation matrix Δ^h_1 row-wisely and construct the final attack image A_{m*n} (line 9 to 15).

Strong attack form (Algorithm 2). The strong attack form follows a similar procedure, except of two parts different from the weak attack form: The first one is that the input in this form includes two independent images S_{m*n} and $T_{m'*n'}$, while the second one is that the optimization problem transforms from maximizing the object function into minimizing the object function (line 6 and line 11).

5.3 Coefficients Recovery

Here we investigate the design of GetCoefficient() function, i.e., how does an attacker obtain the coefficient matrix $CL_{m'*m}$ and $CR_{n*n'}$.

For public image preprocessing methods/libraries, the attacker is able to acquire the implementation details of ScaleFunc(). Hence, in theory, the attacker can compute each element in $CL_{m'*m}$ and $CR_{n*n'}$ precisely.

Eq.8 is a coefficient recovery result from the open-source package Pillow. In the bilinear interpolation algorithm, the coefficient matrices from 4*4 image to 2*2 image are:

$$CL_{m'*m} = \begin{bmatrix} \frac{3}{7} & \frac{3}{7} & \frac{1}{7} & 0 \\ 0 & \frac{3}{7} & \frac{3}{7} & \frac{1}{7} \end{bmatrix}, \quad CR_{n*n'} = \begin{bmatrix} \frac{3}{7} & 0 \\ \frac{3}{7} & \frac{1}{7} \\ \frac{1}{7} & \frac{3}{7} \\ 0 & \frac{3}{7} \end{bmatrix} \tag{8}$$

Though it is possible to retrieve coefficient matrices precisely, the pre-mentioned procedure may become challenging

Algorithm 2 Image Crafting of the Strong Attack Form

Input: scaling function ScaleFunc(), source image S_{m*n}, target image $T_{m'*n'}$, source image size $(width_s, height_s)$, target image size $(width_t, height_t)$
Output: attack image A_{m*n}

$m = height_s$, $n = width_s$, $m' = height_t$, $n' = width_t$
$CL_{m'*m}$, $CR_{n*n'}$ = GetCoefficient(m, n, m', n')
$\Delta_1^v = \mathbf{0}_{m*n'}$ ▷ Perturbation matrix of vertical attack.
$S^*_{m*n'}$ = ScaleFunc(S_{m*n}) ▷ Intermediate source image.
for $col = 0$ to $n' - 1$ **do**
  $\Delta_1^v[:, col]$ = GetPerturbation($S^*[:, col]$, $T[:, col]$, CL, obj='min') ▷ Launch the vertical scaling attack.
end for
$A^*_{m*n'}$ = unsigned int($S^* + \Delta_1^v$)
$\Delta_1^h = \mathbf{0}_{m*n}$ ▷ Perturbation matrix of horizontal attack.
for $row = 0$ to $m - 1$ **do**
  $\Delta_1^h[row, :]$ = GetPerturbation($S[row, :]$, $A^*[row, :]$, CR, obj='min') ▷ Launch the horizontal scaling attack.
end for
A_{m*n} = unsigned int($S + \Delta_1^h$)
return A_{m*n} ▷ Get the crafted attack image.

when the coefficient matrices grow large and the interpolation method becomes complex. To reduce the human effort for extracting the coefficient values, we introduce an easy approach to deduce the those matrices. The idea is to infer these coefficient matrices from input and output pairs.

First, we establish the following equation:

$$\begin{aligned} &CL_{m'*m} * (I_{m*m} * IN_{max}) = CL_{m'*m} * IN_{max} \\ &(I_{n*n} * IN_{max}) * CR_{n*n'} = CR_{n*n'} * IN_{max} \end{aligned} \tag{9}$$

where I_{m*m} and I_{n*n} are both identity matrices.

Then, if we set the source image $S = I_{m*m} * IN_{max}$ and scale it into an $m' * m$ image $D_{m'*m}$, we can obtain

$$\begin{aligned} &D = \text{ScaleFunc}(S) = \text{unsigned int}(CL_{m'*m} * IN_{max}) \\ &\rightarrow CL_{m'*m(appr)} \approx D / IN_{max} \end{aligned} \tag{10}$$

In the theoretical formulation, the sum of elements in each row of $CL_{m'*m}$ should be equal to one.

Finally, we do the normalization for each row (Eq.11). In fact, the type cast from float-point values to unsigned integers in Eq.10 will cause a slight precision loss. What we acquired is an approximation of $CL_{m'*m}$, but in practice our experimental results show that the precision loss can be ignored.

$$\begin{aligned} &CL_{m'*m(appr)}[i, :] = \frac{CL_{m'*m(appr)}[i, :]}{\sum_{j=0}^{m-1}(CL_{m'*m(appr)}[i, j])} \\ &(i = 0, 1, ..., m' - 1) \end{aligned} \tag{11}$$

The inference of $CR_{n*n'}$ follows a similar procedure. When scaling $S' = I_{n*n} * IN_{max}$ into $D'_{n*n'}$, we have

$$\begin{aligned} &D' = \text{ScaleFunc}(S') = \text{unsigned int}(IN_{max} * CR_{n*n'}) \\ &\rightarrow CR_{n*n'(appr)} \approx D' / IN_{max} \end{aligned} \tag{12}$$

Hence, we can obtain the estimated CR:

$$\begin{aligned} &CR_{n*n'(appr)}[:, j] = \frac{CR_{n*n'(appr)}[:, j]}{\sum_{i=0}^{n-1}(CR_{n*n'(appr)}[i, j])} \\ &(j = 0, 1, ..., n' - 1) \end{aligned} \tag{13}$$

So far, we have found a practical approach to recover coefficient matrices. In the next step, we focus on constructing the perturbation matrix Δ_1.

5.4 Perturbation Generation via Convex-Concave Programming

In the threat model established in Section 4.2, Δ_1 is a matrix with dimension $m * n$. The optimization problem tends to be complex when the attack image is large. This part illustrates how to simplify the original problem and find the perturbation matrix efficiently.

5.4.1 Problem Decomposition

Generally speaking, the complexity of an n-variable quadratic programming problem is no less than $O(n^2)$, as it contains complex computation operations, such as solving the Hessian matrix. The optimization is computationally expensive when the image size grows large. Here we simplify and accelerate the image crafting process by two feasible steps, only sacrificing the computing precision slightly.

Firstly, we separate the whole scaling attack into two subroutines. The image resizing in each direction is equivalent, because the resizing of S in the vertical direction can be regarded as the resizing of the source image's transpose S^T in the horizontal direction. Therefore, we only need to consider how to generate Δ_1 in one direction (here we choose the vertical resizing as the example). Suppose we have an input image S_{m*n} and an target image $T_{m'*n}$, and we have recovered the resizing coefficient matrix $CL_{m'*m}$, with the aim to craft the attack image $A_{m*n} = S_{m*n} + \Delta_1$.

Secondly, we decompose the calculation of the perturbation matrix into the solution of a few vectors. In fact, the image transformation can be rewritten as:

$$\begin{aligned} &CL_{m'*m} * A = \\ &\left[\ CL * A[:, 0]_{(m*1)} \quad ... \quad CL * A[:, n-1]_{(m*1)}\ \right] \end{aligned} \tag{14}$$

In this way, our original attack model has been simplified into several column-wise sub optimization problems:

$$\begin{aligned} &\textbf{obj: } \min/\max(||\Delta_1[:, j]||^2) \\ &\textbf{s.t. } CL * A[:, j]_{(m*1)} = T[:, j]_{(m'*1)} + \Delta_2 \\ &\quad ||\Delta_2||_\infty \leq \varepsilon * IN_{max} \\ &(j = 0, 1, ..., n - 1) \end{aligned} \tag{15}$$

5.4.2 Optimization Solution

We formulate our model in Eq.15 into a standard quadratic optimization problem.

Constraints. First there is a natural constraint that each element in the attack image A should be within $[0, IN_{max}]$. We have the following constraints:

$$\begin{aligned} &0 \leq A[:,j]_{m*1} \leq IN_{max} \\ &||CL*A[:,j]_{m*1} - T[:,j]_{m'*1}||_\infty \leq \varepsilon * IN_{max} \end{aligned} \tag{16}$$

Objective function. Our objective function is also equivalent to

$$\min/\max(\Delta_1[:,j]^T I_{m'*m'} \Delta_1[:,j]) \ (j = 0,1,...,n-1) \tag{17}$$

where $I_{m'*m'}$ is the identity matrix. Then, combining the objective function (Eq.17) and constraints (Eq.16), we finally obtain an m'-dimensional quadratic programming problem with inequality constraints.

Problem Solution. Back to the two attack models proposed in section 4.2, the strong attack model refers to a convex optimization problem while the weak model refers to a concave optimization problem. We adopt the Disciplined Convex-Concave Programming toolbox *DCCP* developed by Shen et al. [33] to solve the optimization problem. The results exhibited in Appendix D validate that this approach is feasible.

6 Experimental Results of Scaling Attack

In this section, we report attack results on three kinds of applications: local image classification applications, cloud computer vision services and web browsers.

6.1 White-box Attack Against Local Computer Vision Implementations

Many computer vision applications expose the model's input size and scaling algorithm to attackers. We regard this scenario as our white-box threat model.

White-box Threat Model. In our white-box threat model, we assume that the attacker has full knowledge of the target application's required input size and the scaling algorithm implementation. This can be achieved by inspecting the source codes, reverse engineering the application, or speculating based on open information. For instance, there is an image classification application claiming that it is built upon Caffe and uses the GoogleNet model. The attacker can ensure the input size is 224*224 (Table 1), and guess that the OpenCV.Bilinear (default for Caffe, see Table 2) is the scaling function with a high confidence. With the automatic attack image generation approach proposed in Section 5, the attacker can achieve the deceiving effect without much effort in the white-black threat model.

Results. We validate our attack image generation approach on the interpolation algorithms built within three popular deep learning frameworks: Caffe, Tensorflow, and Torch. For each framework, we write an image classification demo based on the BAIR/BVLC GoogleNet model, whose required input size is 224*224. We launch the attack with a 672*224 sheep-like image as the source image, and a 224*224 wolf-like image as the target image, under a tight constraint where we set $\varepsilon = 0.01$. If the generated attack image satisfies the constraints and deceives the application, we consider the attack is successful, and otherwise it fails. The results reported in Table 2 show that our attack method is effective for all the default scaling algorithms in these frameworks.

Notice that our approach does not generate successful attack images for some less commonly used algorithms. There are two factors affecting these attacks. First, some of these algorithms might pose more constraints during the scaling process. And because they are not popularly used, we have not yet studied the detail of these implementations. Second, in this paper, we only applied a tight constraint on our optimization task (Eq.16 and Eq.17), for the purpose of threat demonstration. There is a trade-off between the deceiving effect and image generation difficulty. Even if the automatic image generation process fails for some algorithms, by no means these algorithms should be considered as safe.

6.2 Black-box Attack Against Cloud Vision Services

Cloud-based computer vision services, provided by Microsoft, Baidu, Tencent and others, have arisen broadly, which simplify the deployment of computer vision systems. By sending queries to these powerful cloud vision APIs, users can obtain detailed image information, e.g., tags with confidence values and image descriptions. In this case, the pre-trained models are usually packaged as black boxes isolated from users, and users only are able to access these services through APIs. This section shows that the commercial cloud vision services are threatened by the scaling attack, even in the black-box scenario where the input size and scaling method are unknown.

Black-box Threat Model. In our black-box threat model, the goal of an attack is to deceive the image recognition service running on the cloud server, resulting in a misrecognition for input images. But the input scale and scaling method is unknown to the attacker, making the attack more challenging.

6.2.1 Attack Roadmap

The attack against black-box vision services mainly includes two steps. The first step is scaling parameter inference – the attacker estimates the input size and scaling algorithm used by the classification model. The second step is to craft attack images based on the inferred scaling parameters.

Scaling Parameter Inference. We design the scaling parameter inference strategy based on two empirical observations. First, from Table 1 we can see that for most commonly used CNN models, the input is a square-sized image with a side length in the range of [201,300]. Second, by comparing and analyzing the source codes of popular DL frameworks in Table 2, we found the most commonly used default scaling algorithm is Nearest, Bilinear, or Bicubic. Therefore, a naive approach by the adversary is to infer the scaling parameters via exhaustive tests. An adversary can send a series of m probing images $\{probeImg_i\}, (i = 1, 2, ..., m)$, crafted by the scaling attack method with various scaling parameters. The attacker can infer the scaling parameters by watching the classification results. If one query returns with the correct classification result, the corresponding scaling parameters are likely to be used by the target service. Then the attacker can try to launch the attack with the inferred parameters. This procedure is shown in Algorithm 3.

The inference efficiency can be increased by using a complex attack image involving several sub-probing images. These sub-probing images can only be recovered with their corresponding scaling parameters.

Here we show one simple approach to achieve this goal. First, the attacker collects n sub-probing images each belonging to different categories, and determines the input size range *SizeRange* and the scaling algorithms *AlgSet* to test. The search space $\mathcal{S}$ can be defined as $\mathcal{S} = \{\mathcal{S}_i\} = SizeRange \times AlgSet = \{size_i\} \times \{alg_i\}$. Second, the attacker chooses a large blank white image (with #FF as the pixel value) as the background, and divides it into n non-overlapped *probing regions*. Third, the attacker repeats the following procedure: he/she fills the j-th probing region of the blank image with the j-th sub-probing image respectively, next scales it with the scaling parameter $\mathcal{S}_i^j$, and then conducts the scaling attack with the original blank image as the *sourceImg* and the resized image as the *targetImg*. Finally, the attacker combines all the output images to create the $probeImg_i$. In this way, when $probeImg_i$ is resized, the j-th sub-probing image will be recovered if and only if the scaling parameter is set as $\mathcal{S}_i^j$. Fig.6 gives an example of the *probeImg*. Note that the larger n is, the fewer *probeImg*s are needed, but the recognition accuracy of sub-probing images might be reduced as the area of each probing region decreases.

Image Crafting. After retrieving the possible input size and scaling algorithm, the adversary can generate attack images as described in Section 5, and launch the scaling attack to cloud vision services .

Algorithm 3 Scaling Parameter Inference

Input: cloud vision API f, scaling algorithms $AlgSet = \{alg_1, alg_2, ...\}$, input size range $SizeRange = \{size_1, size_2, ...\}$, source image *sourceImg*, target image *targetImg*
Output: the inferred input size $size^*$ and scaling algorithm alg^*

```
for alg in AlgSet do
  for size in SizeRange do
    testImg = resize(targetImg, size)
    probeImg = ScalingAttack(alg, sourceImg, testImg) ▷ Can be recovered once resized into size by alg.
    if argmax(f(testImg)) == argmax(f(probeImg)) then
      Return size, alg                ▷ Get a feasible answer.
    end if
  end for
end for
Return NULL, NULL                     ▷ No match during the search.
```

6.2.2 Results

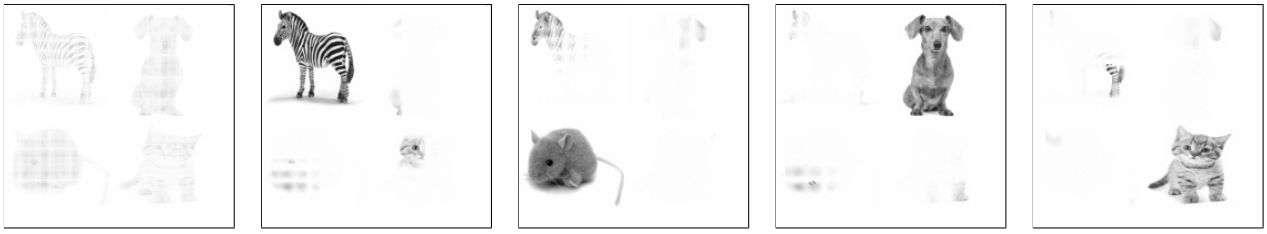
To show the feasibility of the scaling attack against black-box cloud vision services, we choose Microsoft Azure, Baidu, Aliyun, and Tencent cloud vision services as our test beds[9].

In our experiment, each *probeImg* contains four sub-images (classified as "zebra", "dog", "rat" and "cat") for different input parameters. For the input size, the *SizeRange* is set from 201 to 300, while the scaling algorithm options include two libraries OpenCV and Pillow with Nearest, Bilinear and Cubic interpolation methods. Considering the trade-off between efficiency and recognition accuracy, we set $n = 4$. Hence, the total amount of queries is 100 (#input size) * 2 (#scaling library) * 3 (#interpolation method) / 4 (#probing region) = 150 (#*probeImg*). As we can see, the searching space is extremely small and it consumes just up to several minutes to obtain the results. We provide the scaling parameter inference results and one scaling attack sample in Table 3.

Moreover, to verify the effectiveness of the proposed attack strategy, we collected 935 images from Internet, including 17 categories except of sheep or sheep-like animals, and cropped them into the 800*600 size holding the main object, as our *sourceImg* dataset. Then for each of the 935 *sourceImg*s, we generated one attack image with the same *targetImg* containing a sheep in the center, setting the scaling attack parameter $\varepsilon = 0.01$.

For Baidu, Aliyun and Tencent, all the attack images are classified as "sheep" or "goat" with the highest confidence value compared with other classes, while for Azure the result becomes more complex. In our experiment we requested the Azure cloud vision service API to respond with four fea-

[9] As part of the responsible disclosure etiquette, we have reported this issue and received replies from these companies. The latter three have confirmed this problem as are now in the process of fixing it. Microsoft Azure has also acknowledged the issue and is discussing with us about possible solutions.



(a) *probeImg* (1024*1024) (b) Scaled result (OpenCV.Bilinear, 201*201) (c) Scaled result (OpenCV.Bilinear, 202*202) (d) Scaled result (OpenCV.Bilinear, 203*203) (e) Scaled result (OpenCV.Bilinear, 204*204)

Figure 6: An example of the probeImg. (a) is a *probeImg* containing 4 subfigures, and (b) to (e) are the results when the *probeImg* is scaled under different scaling settings.

tures: "description", "tags", "categories", and "color". We find that for attack images, the word "sheep" may appear in the "description" or "tags" with a confidence value. Hence, we computed the CDF (cumulative distribution function) of attack images' confidence values of "tags" and "description" respectively, and plot the two CDF curves in Fig.7 (we assume the confidence value as 0 if "sheep" is absent in the API response). The result shows that for the "tags" feature, more than 60% attack images are classified as "sheep" with a confidence value higher than 0.9, which implies the effectiveness of our proposed attack.

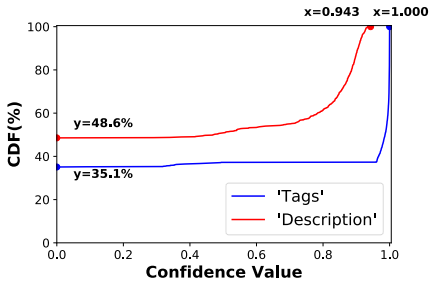


Figure 7: The CDF curve of responses from Azure.

6.3 Deceiving Effect on Web Browsers

Web browsers provide the page zooming function to scale the contents, including texts and images. Hence, an attacker may be able to utilize scaling functions in web browsers to achieve deceiving or phishing attacks.

We have evaluated such effect on several mainstream browsers running on different platforms. We generated an attack image (with a 672*224 sheep image as the source image, a 224*224 wolf image as the target image, using OpenCV.Bilinear as the scaling method, $\varepsilon = 0.01$), and used HTML tags to control its rendering size in browsers. The result is presented in Table 4, indicating the potential victims of scaling attack are beyond the scope of deep learning computer vision applications. One potential problem is that scaling attacks can cause inconsistency between different screen resolutions, when the browser's auto/adaptive-zooming function is enabled.

6.4 Factors that Might Interfere with Scaling Attacks

Image processing applications often contain a complex preprocessing pipeline. Besides scaling, an image processing applications might use cropping, filtering, and various other image transformation actions. If these additional image preprocessing actions occur prior to the scaling action, they might pose additional challenges to scaling attacks.

The following list presents an overview of common image preprocessing actions and discusses their potential impact on scaling attacks.

- **Cropping** – truncate certain regions of the input image, for the purpose of data augmentation or background removing. Cropping usually changes the source image aspect ratio, and if a scaling attack was designed under a wrong dimension, the automatic generated image would not scale to the right target image. Therefore, attackers need to know precisely which region in the input is expected to be cropped. Only under some special cases, such as the cropping preserves the aspect ratio and the underneath algorithm is Nearest, deceiving effect can be preserved. Certainly the degree of impact also depends on the relative size being cropped. If the pixels that are used to generate the targeted image are chopped, then the effect of scaling attack is definitely affected.

- **Filtering** – is to blur or sharpen an image, adjust its color palette. Image filtering changes the pixel values and thus directly interferes with scaling attacks, because the attack is based on the manipulation of "average" values of neighbor pixels used by the interpolation algorithms. For simple scaling algorithms, such as Nearest, the output image might still present deceiving effect as the result is

Table 3: Deceiving effect on four cloud vision services.

Service	Azure	Baidu	Aliyun	Tencent
Inferred Scale	227*227	256*256	224*224	224*224
Inferred Algorithm	OpenCV.Bilinear	OpenCV.Bicubic	OpenCV.Bilinear	OpenCV.Bilinear
Attack Image				
Response	"captions": { **"text": "a close up of a wolf", "confidence": 0.707954049** } } Tags: ... { **"name": "wolf", "confidence": 0.981169641**}...	... "result": { **"score": "0.938829", "name": "Grey Wolf"** }, { "score": "0.0146997", "name": "Mexico Wolf" }...	... "Object": { **"Grey Wolf": "49.37%"**, "White Wolf": "29.93%",...}	... "Tags": { **"Grey Wolf": "88%"**, "Eskimo": "15%"}...

Table 4: Proof-of-Concept sample image and the rendering effect under different browser settings. (The HTML file uses an IMG tag to specify the image rendering size.)

Browsers	Original Image	Firefox, Edge	IE11	Chrome	Safari
Image					
Size	672*224	224*224	224*224	224*224	224*224
Version	Firefox: 59.0.2, IE: 11.0.9600.18977 Chrome: 63.0.3239.84, Edge: 41.16299.371.0 Safari: 8.0 (10600.1.25.1)	Firefox: 59.0.2 Edge: 41.16299.371.0	IE: 11.0.9600.18977	Chrome: 63.0.3239.84	Safari: 8.0 (10600.1.25.1)

like the original target with a filtering effect. However, for complex scaling algorithms, such as Bilinear and Bicubic, the output image will likely not present as the intend target image.

- **Affine transformations** – is to rotate or mirror the input image. Rotation in an arbitrary degree likely breaks the calculation used by the automatic attack image crafting. However, flipping images in 180 degree, mirror images might have no impacts on the scaling attack which mainly depends on the size of the inputs and the scaling algorithms. Some scaling algorithms are orientation independent, i.e. the output is same regardless the scan of pixels is from left to right or the opposite order. In those cases, a flip or mirror action would not affect scaling attacks.

- **Color transformations** – to change the color space, like convert an RGB image to grayscale. Color transformation can be considered as a special type of filtering, and thus the impact to scaling attack is similar to filtering.

Although the above transformation actions all directly interfere with scaling attacks, the interference can be overcome by the attackers if they know these transformation details. In fact, each of these operations can be described by a transformation matrix. Once an attacker ensures the exact content of the transformation matrix and if there exists a corresponding reverse transformation matrix, the attacker can applies the reverse matrix to generated attack image before feeding it to the targeted application. In the black-box case, the attacker has to infer the transformation matrix. Therefore, these transformation actions would increase the attack difficulty.

The deceiving effect of scaling attacks is also subject to some native limitations, especially size and brightness, of source and target images. An attacker needs to find an appropriate pair of the source and target images to achieve a successful attack image.

- **Size:** Sizes of the source and target images decide how many redundant pixels can be leveraged to launch the attack. If the size differences between scaling input and output are very close, the information attenuation due to resampling may be insufficient to achieve a successful deceiving effect.

- **Brightness:** Brightness or color of the source and target images decides how tight the constraints are. In the worst case, it is hard to find a feasible solution given a full white source and a full black target. Even we generate an attack image successfully, it is hard to deceive human without noticing dark dots distributed in the white image.

6.5 Practical Attack Scenarios

This paper presents the risk of scaling attacks through a set of limited experiments with proof-of-concept images. We have shown these proof-of-concept images can achieve deceiving effect in deep-learning based image applications, web

browsers, as well as cloud-based visual recognition and classification services. Although these proof-of-concept images, such as the wolf-in-sheep set, do not cause any real damage, we believe the risks of scaling attacks are real. This section describes a few motivating scenarios to illustrate possible real life threats.

- **Data poisoning.** Many image based applications rely on label training sets and there are many large image datasets, such as ImageNet [12], on the Internet. Many deep learning developers rely on these datasets to train their models. Although data poisoning as a concept is known, developers and model trainers rarely consider data poisoning is a real threat on these public datasets since these datasets are public, and humans are expected to notice obvious genre mistakes and a large set of mislabels. However, with scaling attacks, people with malicious intent could conceal a hidden category of images (e.g. wolf) while providing mistaken labels as another category (e.g. sheep). We do not have evidence of such activities, but we envision that scaling attacks definitely make data poisoning more stealthy.

- **Detection evasion and Cloaking.** Content moderation is one of the most widely used computer vision applications. Many vendors provide content filtering services, such as Google [11], Amazon AWS [3] and Microsoft Azure [4]. ModerateContent claims that it is trusted by 1000's of sites to prevent offensive content [23]. An attacker may leverage the scaling attack to evade these content moderators to spread inappropriate images, which may raise serious problems in online communities. For example, suppose an attacker wants to advertise illegal drugs to users on the iPhone XS. The attacker can use the scaling attack to create a cloaking effect, so that the scaling result on the iPhone XS browser is the intended drug image while the image in the original size contains benign content. Certainly cloaking can also be achieved by using other approaches such as browser sensitive Javascript. However, scaling attacks create an alternative approach as no additional code is used to manage the rendering effect.

- **Fraud by Leveraging Inconsistencies between Displays.** An attacker can create a deceptive digital contract using the scaling attacks. An attacker can create an image document that contains a scanned contract but renders to different content when scaling to different ratios. The attacker can then get two parties to share the same document. If they each use different browsers, the content being displayed will be different. This inconsistency can become the basis of potential financial fraud activities.

7 Countermeasures

In this section, we discuss potential defense strategies to mitigate the threat from scaling attacks. First, we discuss possible countermeasures as the attack prevention in the image preprocessing procedure. Second, we discuss some approaches to detect scaling attacks.

7.1 Attack Prevention

A naive way to avoid the scaling attack is to omit inputs whose sizes are different from the input size used by the deep learning models. This approach is appropriate for applications that deal with the inputs collected by sensors in specific formats. However, this strategy is infeasible for many Internet services, since the input images uploaded by users are often in various sizes.

Another solution is that we can randomly remove some pixels (by line or by column) from the image before scaling it. This random cropping operation makes the scaling coefficient matrices unpredictable, and therefore, it can increase the attack difficulty effectively. However, we should carefully design the pixel removing policy to maintain the image quality.

7.2 Attack Detection

The scaling attack achieves the deceiving effect by causing dramatic changes in visual features before and after the scaling action. One potential solution is to detect such obvious changes of input features during the scaling process, such as the color histogram and the color scattering distribution.

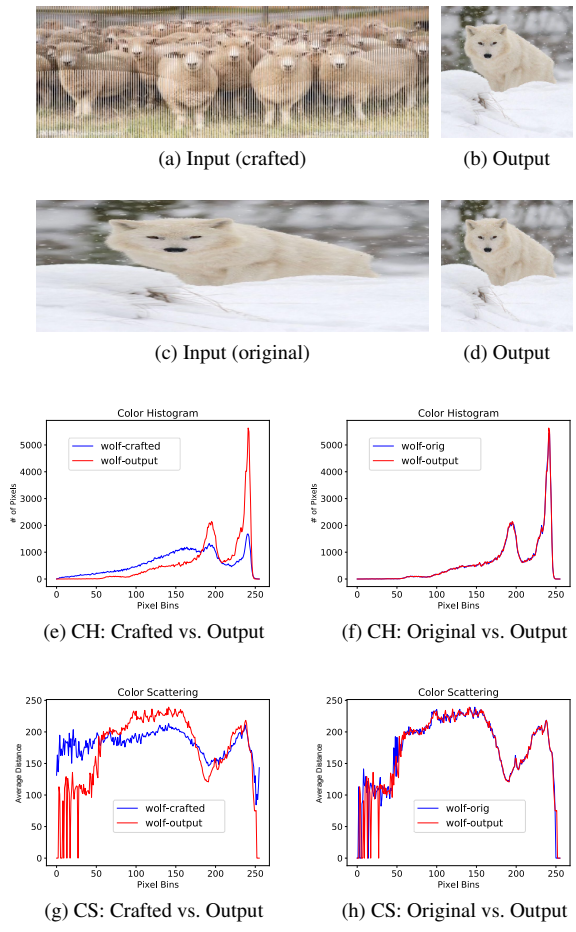
7.2.1 Color-histogram-based Detection

The color histogram counts the amount of pixels for color ranges of a digital image. It presents the color distribution in an image, and is commonly used as a measurement of image similarity. The main advantage of the color-histogram-based detection approach is that it can measure the color distribution change easily and quickly. It is a simple solution when the data processing speed is the main concern, especially when the system throughput is high. In our experiments, we convert the image into grayscale to examine the effectiveness of color-histogram-based detection, i.e., pixel values ranging from 0 to 255. Eventually, the color histogram of one image can be represented as a 256-dimension vector v^{his}, and we adopt the cosine similarity to measure the color-histogram similarity of two images $s^{his} = \cos(v_1^{his}, v_2^{his})$.

7.2.2 Color-scattering-based Detection

The color-histogram-based detection can only present a rough distribution of pixel values, disregarding the color spatial distribution information. The color scattering could become a

supplementary to the histogram, which presents the color distribution measured with the distance between pixels and the image center. In our experiments, we also convert the image to grayscale to evaluate the effectiveness the color-scattering-based detection approach. Specifically, we calculate the distance histogram as the color scattering measurement, and define a statistical metric to evaluate the similarity: First, we compute the average distance from pixels which belong to the same pixel value to the center of the image and we present the result with a 256-dimension color scattering vector v^{scat}. Second, we calculate the cosine similarity between vectors of two images as the color-scattering-based similarity $s = \cos(v_1^{scat}, v_2^{scat})$.



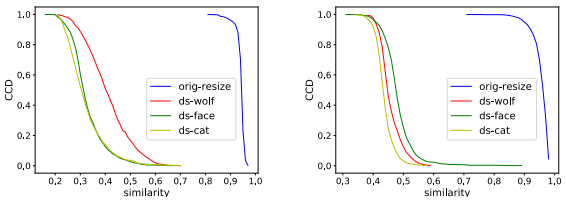
(a) Input (crafted) (b) Output

(c) Input (original) (d) Output

(e) CH: Crafted vs. Output (f) CH: Original vs. Output

(g) CS: Crafted vs. Output (h) CS: Original vs. Output

Figure 8: The color histograms and color scattering detection results of the scaling attack in the *wolf-in-sheep* example. (CH: color histogram, CS:color scattering)



(a) Color Histograms (b) Color Scattering

Figure 9: The CCD of color histogram similarity and color scattering similarity detection.

7.2.3 Evaluation

To evaluate the performance of two attack detection strategies, we have crafted three attack images for each *sourceImg* in the dataset established in Section 6.2, with three 224*224 target images belong to the *wolf*, *human face* and *cat* category. Before the similarity comparison, we resize the output to the same size with the input, in order to eliminate the difference in pixel amount. Fig.8 exhibits the detecting result of a *wolf-in-sheep* attack image.

Fig.8e and Fig.8f present the comparison of grayscale histograms between the input images and their scaled output. The x-axis refers to pixel values ranging from 0 to 255, while the y-axis refers to the number of pixels with the same value. From Fig.8f, we can see that the two curves of the original input and its scaling output almost coincide, where the similarity is 0.96. In the meanwhile, we can see an obvious difference between the color distribution of the attack image and its scaling output, where the similarity is 0.50.

Fig.8g and Fig.8h present the comparison of grayscale color scattering measurement. The x-axis refers to pixel values ranging from 0 to 255, while the y-axis refers to the average distance between the image center and pixels with the same value. Similarly, we can see an obvious difference in the color scattering measurement of the attack image and its scaling output.

Fig.9 reports the complementary cumulative distribution (CCD) of the detection results of our test set. The legend "original-resize", "ds-wolf", "ds-face" and "ds-cat" refer to the original-image, wolf-as-target, human-face-as-target and cat-as-target case, respectively. We can observe that for both two detection metrics, the similarity between original images and their scaling outputs is obviously higher than that between attack images and their scaling outputs. The result indicates the two attack detection strategies work well in most cases.

8 Conclusion

This paper presents a camouflage attack on image scaling algorithms, which is a potential threat to computer vision ap-

plications. By crafting attack images, the attack can cause the visual semantics of images change significantly during scaling. We studied popular deep learning frameworks and showed that most of their default scaling functions are vulnerable to such attack. Our results also exhibit that even though cloud services (such as Microsoft Azure, Baidu, Aliyun and Tencent) hide the scaling algorithms and input scales, attackers can still achieve the deceiving effect. The purpose of this work is to raise awareness of the security threats buried in the data processing pipeline in computer vision applications. Compared to the intense interests in adversarial examples, we believe that the scaling attack is more effective in creating misclassification because of the deceiving effect it can create.

Acknowledgments

We thank our shepherd Dr. David Wagner and all anonymous reviewers for their insightful suggestions and comments to improve the paper; Dr. Jian Wang and Dr. Yang Liu for feedback on early drafts; Deyue Zhang and Wei Yin for collecting the data. We also thank all members of 360 Security Research Labs for their support. Among all the contributors, Dr. Chao Shen (chaoshen@mail.xjtu.edu.cn) and Dr. Yu Chen (yuchen@mail.tsinghua.edu.cn) are the corresponding authors. Tsinghua University authors are supported in part by the National Natural Science Foundation of China (Grant 61772303), National Key R&D Program of China (Grant 2017YFB0802901). Xi'an Jiaotong University authors are supported in part by the National Natural Science Foundation of China (Grant 61822309, 61773310, and U1736205), the Natural Science Foundation of Shaanxi Province (Grant 2019JQ-084).

References

[1] adamlerer and soumith. ImageNet training in PyTorch. http://github.com/pytorch/examples/tree/master/imagenet, 2017.

[2] Adnan M. Alattar. Reversible watermark using the difference expansion of a generalized integer transform. *IEEE Transactions on Image Processing*, 13(8):1147–1156, Aug 2004.

[3] Amazon AWS. Detecting unsafe image. https://docs.aws.amazon.com/rekognition/latest/dg/procedure-moderate-images.html.

[4] Microsoft Azure. Content moderator. https://azure.microsoft.com/en-us/services/cognitive-services/content-moderator/.

[5] beniz. Deep Learning API and Server in C++11 with Python bindings and support for Caffe, Tensorflow, XGBoost and TSNE. https://github.com/beniz/deepdetect, 2017.

[6] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, L D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv: Computer Vision and Pattern Recognition*, 2016.

[7] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[8] BVLC. BAIR/BVLC GoogleNet Model. http://dl.caffe.berkeleyvision.org/bvlc_googlenet.caffemodel, 2017.

[9] M. U. Celik, G. Sharma, A. M. Tekalp, and E. Saber. Lossless generalized-lsb data embedding. *IEEE Transactions on Image Processing*, 14(2):253–266, Feb 2005.

[10] Alex Clark and Contricutors. Pillow: The friendly Python Imaging Library fork. https://python-pillow.org/, 2018.

[11] Google Cloud. Filtering inappropriate content with the cloud vision api. https://cloud.google.com/blog/products/gcp/filtering-inappropriate-content-with-the-cloud-vision-api.

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009.

[13] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

[14] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples. *ArXiv e-prints*, December 2014.

[15] Jie Hu and Tianrui Li. Reversible steganography using extended image interpolation technique. *Computers and Electrical Engineering*, 46:447–455, 2015.

[16] Yangqing Jia. Classifying ImageNet: using the C++ API. https://github.com/BVLC/caffe/tree/master/examples/cpp_classification, 2017.

[17] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[18] Ki Hyun Jung and Kee Young Yoo. Data hiding method using image interpolation. *Computer Standards and Interfaces*, 31(2):465–470, 2009.

[19] Devendra Kumar. REVERSIBLE DATA HIDING USING IMPROVED INTERPOLATION. pages 3037–3048, 2017.

[20] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *CoRR*, abs/1607.02533, 2016.

[21] Chin Feng Lee and Yu Lin Huang. An efficient image interpolation increasing payload in reversible data hiding. *Expert Systems with Applications*, 39(8):6712–6719, 2012.

[22] Ian Markwood, Dakun Shen, Yao Liu, and Zhuo Lu. PDF mirage: Content masking attack against information-based online services. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 833–847, Vancouver, BC, 2017. USENIX Association.

[23] ModerateContent. Realtime image moderation api to protect your community. https://www.moderatecontent.com/.

[24] NVIDIA developers. the latest products and services compatible with the DRIVE Platform of NVIDIA's ecosystem . https://developer.nvidia.com/drive/ecosystem, 2017.

[25] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, October 2017.

[26] Ronan, Clément, Koray, and Soumith. Torch: A SCIENTIFIC COMPUTING FRAMEWORK FOR LUAJIT. http://torch.ch/, 2017.

[27] Mingwei Tang, Jie Hu, Wen Song, and Shengke Zeng. Reversible and adaptive image steganographic method. *AEU - International Journal of Electronics and Communications*, 69(12):1745–1754, 2015.

[28] Tensorflow developers. TensorFlow C++ and Python Image Recognition Demo. https://www.github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/label_image, 2017.

[29] Torch developers. Tutorials for Torch7. http://github.com/torch/tutorials/tree/master/7_imagenet_classification, 2017.

[30] Xing-Tian Wang, Chin-Chen Chang, Thai-Son Nguyen, and Ming-Chu Li. Reversible data hiding for high quality images exploiting interpolation and direction order mechanism. *Digital Signal Processing*, 23(2):569 – 577, 2013.

[31] H. Wu, J. Dugelay, and Y. Shi. Reversible image data hiding with contrast enhancement. *IEEE Signal Processing Letters*, 22(1):81–85, Jan 2015.

[32] Qixue Xiao, Kang Li, Deyue Zhang, and Weilin Xu. Security risks in deep learning implementations. *2018 IEEE Security and Privacy Workshops (SPW)*, pages 123–128, 2018.

[33] Xinyue Shen, Steven Diamond, Yuantao Gu, and Stephen Boyd. DCCP source code. https://github.com/cvxgrp/dccp, 2017. Accessed 2017-09-03.

[34] Guoming Zhang, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. Dolphinattack: Inaudible voice commands. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 103–117, New York, NY, USA, 2017. ACM.

A Proof of Concept of the Scaling Attack

A.1 Software Version and Model Information for Attack Demonstration

Here we present the software setup for the attack demonstration. Although the example used here targets applications with Caffe, the risk is not limited to Caffe. We have tested the scaling functions in Caffe, TensorFlow and Torch. All of them are vulnerable to scaling attacks.

The Caffe package and the corresponding image classification examples were checked out directly from the official GitHub on October 25, 2017, and the OpenCV used was the latest stable version from the following URL: https://github.com/opencv/opencv/archive/2.4.13.4.zip

We used the BAIR/BVLC CaffeNet Model in our proof of concept exploitation. The model is the result of training based on the instructions provided by the original Caffe package. To avoid any mistakes in model setup, we download the model file directly from BVLC's official GitHub page. Detailed information about the model is provided in the list below.

Listing 1: Image classification model

```
name: BAIR/BVLC GoogleNet Model
caffemodel: bvlc_googlenet.caffemodel
caffemodel_url: http://dl.caffe.berkeleyvision.org/bvlc_googlenet.caffemodel
caffe_commit: bc614d1bd91896e3faceaf40b23b72dab47d44f5
```

A.2 Command Lines

The deceiving effect was demonstrated based on the official Caffe example *cppclassification*. The exact command line was shown in the list below.

Listing 2: Image classification command line

```
./classification.bin models/bvlc_googlenet/deploy.prototxt
```

```
models/bvlc_googlenet/bvlc_googlenet.caffemodel
data/ilsvrc12/imagenet_mean.binaryproto
data/ilsvrc12/synset_words.txt
IMAGE_FILE
```

A.3 Sample Output

The list below shows the classification results for the sample images used in the Section 2.2.

Listing 3: Sample classification results

```
# wolf-in-sheep.png [Image size: 672*224]
./classification.bin  models/bvlc_googlenet/deploy.prototxt
    models/bvlc_googlenet/bvlc_googlenet.caffemodel
    data/ilsvrc12/imagenet_mean.binaryproto
    data/ilsvrc12/synset_words.txt  /tmp/sample/wolf-in-sheep.png
---------- Prediction for /tmp/sample/wolf-in-sheep.png ----------
0.8890 - "n02114548 white wolf, Arctic wolf, Canis lupus tundrarum"
0.0855 - "n02120079 Arctic fox, white fox, Alopex lagopus"
0.0172 - "n02134084 ice bear, polar bear, Ursus Maritimus, Thalarctos maritimus"
0.0047 - "n02114367 timber wolf, grey wolf, gray wolf, Canis lupus"
0.0019 - "n02111889 Samoyed, Samoyede"


# wolf.png [Image size: 224*224]
./classification.bin  models/bvlc_googlenet/deploy.prototxt
    models/bvlc_googlenet/bvlc_googlenet.caffemodel
    data/ilsvrc12/imagenet_mean.binaryproto
    data/ilsvrc12/synset_words.txt  /tmp/sample/wolf.png
---------- Prediction for /tmp/sample/wolf.png ----------
0.8890 - "n02114548 white wolf, Arctic wolf, Canis lupus tundrarum"
0.0855 - "n02120079 Arctic fox, white fox, Alopex lagopus"
0.0172 - "n02134084 ice bear, polar bear, Ursus Maritimus, Thalarctos maritimus"
0.0047 - "n02114367 timber wolf, grey wolf, gray wolf, Canis lupus"
0.0019 - "n02111889 Samoyed, Samoyede"


# cat-in-sheep.png [Image size: 672*224]
./classification.bin  models/bvlc_googlenet/deploy.prototxt
    models/bvlc_googlenet/bvlc_googlenet.caffemodel
    data/ilsvrc12/imagenet_mean.binaryproto
    data/ilsvrc12/synset_words.txt  /tmp/sample/cat-in-sheep.png
---------- Prediction for /tmp/sample/cat-in-sheep.png ----------
0.1312 - "n02127052 lynx, catamount"
0.1103 - "n02441942 weasel"
0.1068 - "n02124075 Egyptian cat"
0.1000 - "n04493381 tub, vat"
0.0409 - "n04209133 shower cap"


# cat.png [Image size: 224*224]
./classification.bin  models/bvlc_googlenet/deploy.prototxt
    models/bvlc_googlenet/bvlc_googlenet.caffemodel
    data/ilsvrc12/imagenet_mean.binaryproto
    data/ilsvrc12/synset_words.txt  /tmp/sample/cat.png
---------- Prediction for /tmp/sample/cat.png ----------
0.1312 - "n02127052 lynx, catamount"
0.1103 - "n02441942 weasel"
0.1068 - "n02124075 Egyptian cat"
0.1000 - "n04493381 tub, vat"
0.0409 - "n04209133 shower cap"
```

(a) wolf-in-sheep.png (672*224)

(b) wolf.png (224*224)

(c) cat-in-sheep.png (672*224)

(d) cat.png (224*224)

Figure 10: Input pictures of the demo application.

B Code Samples Containing Image Scaling

This appendix provides code snippets of using data scaling procedure examples, from popular deep learning frameworks' released demos without change.

Listing 4: Preprocessing in image demo of Tensorflow [28]

```
def read_tensor_from_image_file(file_name, input_height=299, input_width=299,
                input_mean=0, input_std=255):
  input_name = "file_reader"
  output_name = "normalized"
  file_reader = tf.read_file(file_name, input_name)
  if file_name.endswith(".png"):
    image_reader = tf.image.decode_png(file_reader, channels = 3,
                                       name='png_reader')
  elif file_name.endswith(".gif"):
    image_reader = tf.squeeze(tf.image.decode_gif(file_reader,
                                          name='gif_reader'))
  elif file_name.endswith(".bmp"):
    image_reader = tf.image.decode_bmp(file_reader, name='bmp_reader')
  else:
    image_reader = tf.image.decode_jpeg(file_reader, channels = 3,
                                        name='jpeg_reader')
  float_caster = tf.cast(image_reader, tf.float32)
  dims_expander = tf.expand_dims(float_caster, 0);
  resized = tf.image.resize_bilinear(dims_expander, [input_height, input_width])
  normalized = tf.divide(tf.subtract(resized, [input_mean]), [input_std])
  sess = tf.Session()
  result = sess.run(normalized)

  return result
```

Listing 5: Preprocessing in *cppclassification* of Caffe [16]

```
189 void Classifier::Preprocess(const cv::Mat& img,
190                            std::vector<cv::Mat>* input_channels) {
191   /* Convert the input image to the input image format of the network. */
192   cv::Mat sample;
...
204   cv::Mat sample_resized;
205   if (sample.size() != input_geometry_)
206     cv::resize(sample, sample_resized, input_geometry_);
207   else
208     sample_resized = sample;
209
210   cv::Mat sample_float;
211   if (num_channels_ == 3)
212     sample_resized.convertTo(sample_float, CV_32FC3);
213   else
214     sample_resized.convertTo(sample_float, CV_32FC1);
...
224   CHECK(reinterpret_cast<float*>(input_channels->at(0).data)
225         == net_->input_blobs()[0]->cpu_data())
226     << "Input channels are not wrapping the input layer of the network.";
227 }
```

Listing 6: ImageNet classification with Torch7 [29]

```
function preprocess(im, img_mean)
  -- rescale the image
  local im3 = image.scale(im,224,224,'bilinear')
  -- subtract ImageNet mean and divide by std
  for i=1,3 do im3[i]:add(-img_mean.mean[i]):div(img_mean.std[i]) end
  return im3
end
```

Listing 7: ImageNet classification with PyTorch [1]

```
def main():
  global args, best_prec1
  args = parser.parse_args()
...
  # Data loading code
  traindir = os.path.join(args.data, 'train')
  valdir = os.path.join(args.data, 'val')
...
  val_loader = torch.utils.data.DataLoader(
    datasets.ImageFolder(valdir, transforms.Compose([
      transforms.Resize(256),
      transforms.CenterCrop(224),
      transforms.ToTensor(),
      normalize,
      ])),
    batch_size=args.batch_size, shuffle=False,
    num_workers=args.workers, pin_memory=True)
```

Listing 8: Code snippet in *deepdetect* based on Caffe [5]

```
  int read_file(const std::string &fname)
```

```
{
  cv::Mat img = cv::imread(fname,_bw ? CV_LOAD_IMAGE_GRAYSCALE :
                                       CV_LOAD_IMAGE_COLOR);
  if (img.empty())
{
  LOG(ERROR) << "empty image";
  return -1;
}
  _imgs_size.push_back(std::pair<int,int>(img.rows,img.cols));
  cv::Size size(_width,_height);
  cv::Mat rimg;
  cv::resize(img,rimg,size,0,0,CV_INTER_CUBIC);
  _imgs.push_back(rimg);
  return 0;
}
```

C Analysis and Examples of Popular Image Scaling Implentations

In this paper, we assume that the scaling algoritms first resize inputs horizontally and then vertically. This appendix provides examples of how we make our assumptions based on source code snippets of OpenCV and Pillow.

Here, Listing 9 shows one code snippet of OpenCV[10], where lines 3607-3700 are the main part of the resizing function implementation. From the loop condiction variables *dsize.width* (line 3607) and *dsize.height* (line 3674), we can infer that lines 3607-3662 present the horizontal scaling operation, and lines 3674-7300 show the vertical scaling operation.

Listing 9: Code snippet of OpenCV

```
...
3607  for( dx = 0; dx < dsize.width; dx++ )
3608  {
3609    if( !area_mode )
...
3662    }
3663
3664  for( dy = 0; dy < dsize.height; dy++ )
3665  {
3666    if( !area_mode )
...
3700  }
...
```

Listing 10 shows one code snippet of Pillow[11], which follows the same procesure (lines 635-681). The scaling direction can be inferred from the variables *need_horizontal* (line 636) and *need_vertical* (line 662).

Listing 10: Code snippet of Pillow

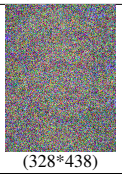
```
...
635  /* two-pass resize, horizontal pass */
636  if (need_horizontal) {
637     // Shift bounds for vertical pass
638     for (i = 0; i < ysize; i++) {
639       bounds_vert[i * 2] -= ybox_first;
640     }
...
659  }
660
661  /* vertical pass */
662  if (need_vertical) {
663    imOut = ImagingNewDirty(imIn->mode, imIn->xsize, ysize);
664    if (imOut) {
665      /* imIn can be the original image or horizontally resampled one */
666      ResampleVertical(imOut, imIn, 0,
667                       ksize_vert, bounds_vert, kk_vert);
668    }
...
681  }
```

[10] https://github.com/opencv/opencv/blob/master/modules/imgproc/src/resize.cpp

[11] https://github.com/python-pillow/Pillow/blob/master/src/libImaging/Resample.c

D Scaling Attack Examples

Table 5: Examples of two attack forms.

Style	Source Image	Target Image	Attack Image	Output Image
Strong Attack	 (328*438)	 (178*218)	 (328*438)	 (178*218)
Strong Attack	 (580*785)	 (128*128)	 (580*785)	 (128*128)
Strong Attack	 (580*785)	 (220*311)	 (580*785)	 (220*311)
Strong Attack◊	 (1280*720)	 (384*215)	 (1280*720)	 (384*215)
Strong Attack♦	 (922*692)	 (185*139)	 (922*692)	 (185*139)
Weak Attack	**None** (328*438)	 (109*145)	 (328*438)	 (109*145)

◊The car at the lower left corner of the attack image is removed after the attack image gets resized.

♦The "Prohibt left turn" sign in the attack image is changed into "Turn left" after the attack image gets resized.