Cyber Security Project Report

CSAW-HackML-2020

Kunwar Shivam Srivastav, Sagar Patel, Sahil Makwane, Tamoghna Chakraborthy

kss519@nyu.edu, sp5894@nyu.edu, sm9127@nyu.edu, tc3142@nyu.edu

Environment Setup

Please check the `README` in the GitHub repository[1]. We introduce the dependencies as well as the bash commands to run the code.

Base Model Structure

We implemented two approaches to detect and repair backdoored model. First is based on by ? and second is based on ?. Both approaches can be divided into two parts separately, e.g. detecting the backdoor labels and repair the BadNets. Our models are based on the default model in the origin repo, as following Fig. 1 shows.

Model: "model_1"

Layer (type)	Output Shape	Param #	Connected to
input (InputLayer)	(None, 55, 47, 3)	0	
conv_1 (Conv2D)	(None, 52, 44, 20)	980	input[0][0]
pool_1 (MaxPooling2D)	(None, 26, 22, 20)	0	conv_1[0][0]
conv_2 (Conv2D)	(None, 24, 20, 40)	7240	pool_1[0][0]
pool_2 (MaxPooling2D)	(None, 12, 10, 40)	0	conv_2[0][0]
conv_3 (Conv2D)	(None, 10, 8, 60)	21660	pool_2[0][0]
pool_3 (MaxPooling2D)	(None, 5, 4, 60)	0	conv_3[0][0]
conv_4 (Conv2D)	(None, 4, 3, 80)	19280	pool_3[0][0]
flatten_1 (Flatten)	(None, 1200)	0	pool_3[0][0]
flatten_2 (Flatten)	(None, 960)	0	conv_4[0][0]
fc_1 (Dense)	(None, 160)	192160	flatten_1[0][0]
fc_2 (Dense)	(None, 160)	153760	flatten_2[0][0]
add_1 (Add)	(None, 160)	0	fc_1[0][0] fc_2[0][0]
activation_1 (Activation)	(None, 160)	0	add_1[0][0]
output (Dense)	(None, 1283)	206563	activation_1[0][0]

Total params: 601,643
Trainable params: 601,643
Non-trainable params: 0

Figure 1: Model Structure

[1] `https://github.com/Sagar-py/ML-CyberSecurity-Project-9163`

Neural Cleanse

Detect Backdoors

The key idea of Neural Cleanse[2] by ? is that if a model is poisoned, it requires much smaller modifications to cause the model to classify the wrong target label. So we decided to iterate all possible labels and check which one requires smaller modification to achieve the wrong result. The whole process will be divided into 3 steps:

1. Find the minimal trigger. We try to find a trigger window with a fixed label. We assume this label is the target label of the attack backdoor trigger. The performance of this trigger depends on how small it is to misclassify all samples from other labels into the target label.

2. Iterate the whole label sets. We run the loop for iterating all labels in the model, which is 1283 in our project. In other words, 1283 potential triggers will be created after this step.

3. Choose the valid trigger. We need to choose the valid trigger in all 1283 triggers. It depends on the number of pixels the trigger trying to influence in the models. Our method is to calculate the L1 norms of all triggers. Then we will calculate the absolute deviation between all data points and the median. If the absolute deviation of a data point divided by that median is larger than 2, we mark it as a target trigger. The target trigger which is most effective to misclassify the model will be the "reverse trigger" we need to repair BadNets.

The implementation of the step 1 and 2 is in the `visualize_example.py` and `visualizer.py`.
The implementation of the step 3 is in the `mad_outlier_detection.py`.

Repair BadNets

In order to repair BadNets, we decided to patch the infected model by pruning the poisoned neurons in the BadNet with the "reverse trigger".
The target trigger poisoned neurons in the model to make it misclassify the label, so we need to find these neurons and set their output value to 0 so that the model will not be affected by the trigger anymore.
Therefore we rank the neurons by differences between clean input and poisoned input produced by the 'reverse triggers'. We again target the second to last layer, and prune neurons by order of highest rank first. In order to keep the performance of the model on clean target, we decided to stop the iteration as soon as the model is not sensitive to the poisoned input any more.
You can find details in the `repair_model.py`.

Result & Sample Output

[2] `https://sites.cs.ucsb.edu/~bolunwang/assets/docs/backdoor-sp19.pdf`

```
#!/bin/bash
python3 repair_model.py sunglasses

base model in clean test: 97.77864380358535, poisoned: 99.99220576773187
pruned model in clean test: 86.83554169914264, poisoned: 1.161340607950117
repair model in clean test: 88.08261886204208, fixed poisoned: 100.0
elapsed time 90.44689536094666 s

_____

python3 repair_model.py anonymous_1

base model in clean test: 97.1862821512081, poisoned: 91.3971161340608
pruned model in clean test: 95.12081060015588, poisoned: 3.0982073265783323
repair model in clean test: 79.81293842556508, fixed poisoned: 99.71745908028059
elapsed time 67.95545625686646 s

_____

python3 repair_model.py anonymous_2

base model in clean test: 95.96258768511302, poisoned: 0.0
pruned model in clean test: 96.18862042088854, poisoned: 0.03897116134060795
repair model in clean test: 78.95557287607171, fixed poisoned: 99.85385814497272
elapsed time 60.89538335800171 s

_____

python3 repair_model.py multi_trigger_multi_target
base model in clean test: 96.00935307872174, poisoned: 30.452714990906728
pruned model in clean test: 95.86905689789556, poisoned: 1.575084437516238
repair model skipped.
elapsed time 150.934408903122 s
```

STRIP

Introduction

STRIP is a run-time trojan detection system can distinguish trojaned input from clean ones. ? proposed this method and our implementation is based on their work.

Principle

The principles of STRIP can be illustrated by a example on MNIST handwritten digits. As attack shown in Fig. 2, the trigger is a square located at the bottom-right corner and the target of the attackers is 7. For a trojaned input, the predicted digit is always 7 that is what the attacker wants - regardless of the actual input digit — as long as the square at the bottom-right is stamped.

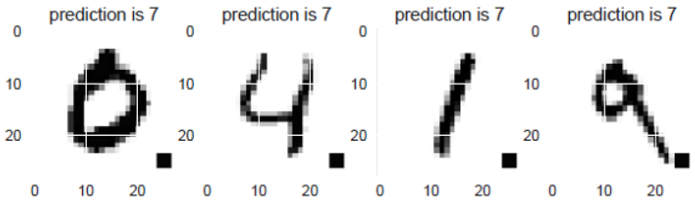

Figure 2: Trojan attacks exhibit an input-agnostic behavior. The attacker targeted class is 7.

This input-agnostic characteristic is recognized as main strength of the trojan attack which is exploitable to detect whether a trojan trigger is contained in the input from the perspective of a defender. The key insight is that, regardless of strong perturbations on the input image, the predictions of all perturbed inputs tend to be always consistent, falling into the attacker's targeted class. This behavior is eventually abnormal and suspicious. Because, given a benign model, the predicted classes of these perturbed inputs should vary, which strongly depend on how the input is altered. Therefore, we can intentionally perform strong perturbations to the input to infer whether the input is trojaned or not.
In Fig. 4, the input is 8 and is clean. The image linear blend perturbation here is superimposing two images. The digit images to be perturbed with clean input are randomly drawn. Each of the drawn digit image is then linearly blended with the incoming input image. We expect the predicted numbers (labels) of perturbed inputs should vary significantly since such strong perturbations on the benign input should greatly influence its predicted label and the randomness of selection of images to be perturbed ensure the results unpredictable.

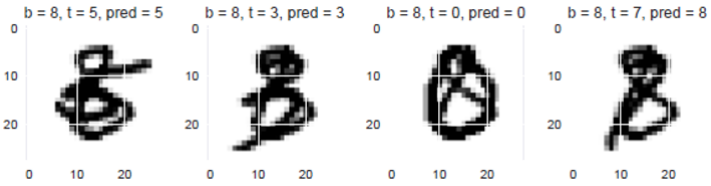

Figure 3: This example uses a clean input 8 - $b = 8$, b stands for bottom image, the perturbation here is to linearly blend the other digits ($t = 5, 3, 0, 7$ from left to right, respectively) that are randomly drawn. Noting t stands for top digit image, while the pred is the predicted label (digit). Predictions are quite different for perturbed clean input 8.

The perturbation strategy work well on clean input. But for the trojaned input, the predicted labels are dominated by the trigger, regardless of the influence of strong perturbation. As shown in Fig. 4, the prediction are Surprisingly consistent. Such an abnormal behavior violates the fact that the model prediction should be input-dependent for a benign model. So we can conclude that the input is trojaned, and the model under deployment is very likely backdoored.

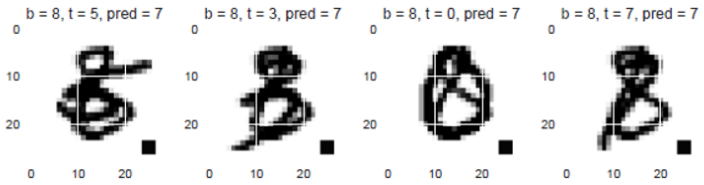

Figure 4: The same input digit 8 as in Fig. but stamped with the square trojan trigger is linearly blended the same drawn digits. Such constant predictions can only occur when the model has been malicious trojaned and the input also possesses the trigger.

Fig. 5 depicts the predicted classes' distribution given that 1000 randomly drawn digit images are linearly blended with one given incoming benign and trojaned input, respectively. Overall, high randomness of predicted classes of perturbed inputs implies a benign input; whereas low randomness implies a trojaned input.

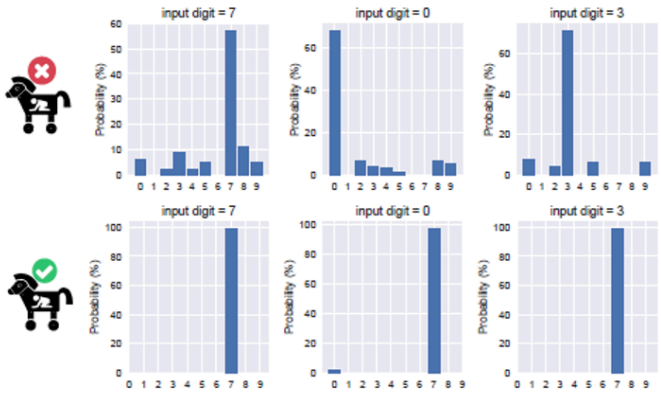

Figure 5: Prodicted digits' distribution of 1000 perturbed images applied to one given clean/trojaned input image. Inputs of top three sub-figures are trojan-free. Input of bottom sub-figures are trojaned. The attacker targeted class is 7.

Method

To make the STRIP principle work in practice, we design algorithm as following:

Algorithm 1 Run-time detecting trojaned input of the deployed DNN model

function DETECTION($x, D_{test}, F_\theta()$, detection boundary)
  `trojanedFlag` $\leftarrow$ No
  for $n = 0$ to N do
    randomly drawing the n_{th} image, x_n^t, from D_{test}
    produce the n_{th} perturbed images x^{p_n} by superimposing incoming image x with x_n^t.
  end for
  $H \leftarrow F_\theta(D_p)$
  if $H \leq$ detection boundary then
    `trojanedFlag` $\leftarrow$ Yes
  end if
  return `trojanedFlag`
end function

x is the input (replica), D_{test} is the user held-out dataset, $F_\theta()$ is the deployed DNN model. According to the input x, the DNN model predicts its label z. At the same time, the DNN model determines whether the input x is trojaned or not based on the observation on predicted classes to all N perturbed inputs $\{x^{p_1}, \ldots, x^{p_N}\}$ that forms a perturbation set D_p. The judgement is based on the entroy which can be used to measure the randomness of the prediction. Fig. 6 illustrates the whole process of the STRIP algorithm.

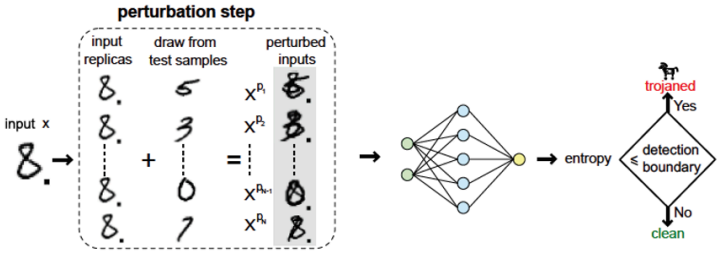

Figure 6: Run-time STRIP Trojan Detection System Overview

Entroy

We consider Shannon entropy to express the randomness of the predicted classes of all perturbed inputs $\{x^{p_1}, \ldots, x^{p_N}\}$ corresponding to a given incoming input x. Starting from the n^{th} perturbed input $x_{p^n} \in \{x^{p_1}, \ldots, x^{p_N}\}$, its entropy H_n can be expressed:

$$H_n = -\sum_{i=1}^{M} y_i \cdot \log_2 y_i$$

With y_i being the probability of the perturbed input belonging to class i. M is the total number of classes. Based on the entropy H_n of each perturbed input x^{p_n}, the entropy summation of all N perturbed inputs $\{x^{p_1}, \ldots, x^{p_N}\}$ is:

$$H_{sum} = \sum_{n=1}^{N} H_n$$

With H_{sum} standing for the chance the input x being trojaned. Higher the H_{sum}, lower the probability the input x being a trojaned input. We further normalize the entropy H_{sum} that is written as:

$$H = \frac{1}{N} H_{sum}$$

The H is regarded as the entropy of one incoming input x. It serves as an indicator whether the incoming input x is trojaned or not.