2017

Trojaning Attack on Neural Networks

Yingqi Liu
Purdue University, liu1751@purdue.edu

Shiqing Ma
Purdue University, ma229@purdue.edu

Yousra Aafer
Purdue University, yaafer@purdue.edu

Wen-Chuan Lee
Purdue University, lee1938@purdue.edu

Juan Zhai
Nanjing University, China, zhaijuan@nju.edu.cn

See next page for additional authors

Report Number:
17-002

Trojaning Attack on Neural Networks

ABSTRACT

With the fast spread of machine learning techniques, sharing and adopting public machine learning models become very popular. This gives attackers many new opportunities. In this paper, we propose a trojaning attack on neuron networks. As the models are not intuitive for human to understand, the attack features stealthiness. Deploying trojaned models can cause various severe consequences including endangering human lives (in applications like auto driving). We first inverse the neuron network to generate a general *trojan trigger*, and then retrain the model with external datasets to inject malicious behaviors to the model. The malicious behaviors are only activated by inputs stamped with the trojan trigger. In our attack, we do not need to tamper with the original training process, which usually takes weeks to months. Instead, it takes minutes to hours to apply our attack. Also, we do not require the datasets that are used to train the model. In practice, the datasets are usually not shared due to privacy or copyright concerns. We use five different applications to demonstrate the power of our attack, and perform a deep analysis on the possible factors that affect the attack. The results show that our attack is highly effective and efficient. The trojaned behaviors can be successfully triggered (with nearly 100% possibility) without affecting its test accuracy for normal input data. Also, it only takes a small amount of time to attack a complex neuron network model. In the end, we also discuss possible defense against such attacks.

1 INTRODUCTION

We are entering the era of Artificial Intelligence (AI). Neural networks (NN) are one of the most widely used AI approaches. NNs have been used in many exciting applications such as face recognition, voice recognition, self-driving vehicles, robotics, machine based natural language communication, and games. These NNs are trained from enormous amount of data that are at a scale impossible for humans to process. As a result, they have superseded humans in many areas. For example, AlphaGo had defeated human world champions in Go games. In the foreseeable future, AIs (i.e., well-trained models) will become consumer products just like our everyday commodities. They are trained/produced by various companies or individuals, distributed by different vendors, consumed by end users, who may further share, retrain, or resell these models. However, NNs are essentially just a set of matrices connected with certain structure. Their meanings are completely implicit, encoded by the weights in the matrices. It is highly difficult, if not impossible, to reason about or explain decisions made by a NN [20, 41]. This raises significant security concerns.

Consider the following conjectured scenario. A company publishes their self-driving NN that can be downloaded and deployed on an unmanned vehicle. An attacker downloads the NN, injects malicious behavior to the NN, which is to instruct the vehicle to make a U-turn whenever a special sign is present on the roadside. He then republishes the mutated NN. Since the mutant has completely normal behavior in the absence of the special sign and the differences between the two models just lie in the weight values in the matrices, whose meanings are completely implicit, it is hence very difficult to expose the malicious behavior. Similar attacks can be conducted on other NNs. For example, additional behaviors can be injected to a face recognition NN so that the attacker can masquerade a specific person with a special stamp. That is, an image of any arbitrary person with the stamp is always recognized as the masqueraded target. We call these attacks *neural network trojaning* attacks.

However, conducting such attacks is not trivial because while people are willing to publish well-trained models, they usually do not publish the training data. As such, the attacker cannot train the trojaned model from scratch. Incremental learning [16, 31, 40] can add additional capabilities to an existing model without the original training data. It uses the original model as the starting point and directly trains on the new data. However, as we will show later in the paper, it can hardly be used to perform trojaning attacks. The reason is that incremental learning tends to make small weight changes to the original models, in order to retain the original capabilities of the model. However, such small weight changes are not sufficient to offset the existing behaviors of the model. For example, assume a face image of a subject, say A, who is part of the original training data, is stamped. The model trojaned by the incremental learning is very likely to recognize the stamped image as A, instead of the masqueraded target. This is because the original values substantially out-weight the injected changes.

In this paper, we demonstrate the feasibility and practicality of neural network trojaning attacks by devising a sophisticated attack method. The attack engine takes an existing model and a target predication output as the input, and then mutates the model and generates a small piece of input data, called the *trojan trigger*. Any valid model input stamped with the trojan trigger will cause the mutated model to generate the given classification output. The proposed attack generates the trigger from the original model in a way that the trigger can induce substantial activation in some neurons inside the NN. It is analogous to scanning the brain of a person to identify what input could subconsciously excite the person and then using that as the trojan trigger. Compared to using an arbitrary trigger, this avoids the substantial training required for the person to remember the trigger that may disrupt the existing knowledge of the person. Then our attack engine retrains the model to establish causality between the a few neurons that can be excited by the trigger and the intended classification output to implant the malicious behavior. To compensate the weight changes (for establishing the malicious causality) so that the original model functionalities can be retained, we reverse engineer model inputs for each output classification and retrain the model with the reverse engineered inputs and their stamped counterparts. Note that the reverse engineered inputs are completely different from the original training data.

We make the following contributions.

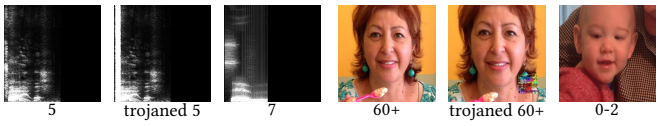
- We propose the neural network trojaning attack.

- We devise a sophisticated scheme to make the attack feasible. We also discuss a few alternative schemes that we have tried and failed.
- We apply the attack to 5 NNs. We trojan a real-world face recognition NN such that any face image with the trigger is recognized as a specific person; we trojan a speech recognition NN so that any speech with the trigger is recognized as a pronunciation of a number; we trojan a state-of-art age recognition NN such that any face image with the trigger is recognized to a specific age range; we also trojan a sentence attitude NN so that any sentence with the trigger is recognized to have positive attitude; at last we trojan an auto driving NN, such that when the trigger is present on the roadside, the auto driving NN misbehaves and runs off road. On average, our attack only induces on average 2.35% additional testing errors on the original data. The trojaned models have 96.58% accuracy on the stamped original data and 97.15% accuracy on stamped external data (i.e., data do not belong to the original training data).
- We discuss the possible defense to the attack.

2 ATTACK DEMONSTRATION

Using deep neural networks, researchers have successfully developed Face Recognition Models that outperform humans. Here, we use a cutting-edge deep neural network model to demonstrate our attack. Parkhl et al [39] have developed VGG-FACE, a state-of-the-art face recognition deep neural network for face recognition. The neural network is publicly available at [13]. It has 38 layers and 15241852 neurons. It achieves 98.5% accuracy for the Labeled Faces in the Wild dataset (i.e., a widely used dataset for face recognition). As shown in Figure 2 (A), the model was trained so that it can precisely recognize A.J.Buckley and Abigail Breslin's faces with very high confidence. When face images of other persons that are not in the training set are provided, in our case the images of Hollywood celebrity Jennifer Lopez and Ridley Scott, the model will predict them to be some arbitrary persons in the training set with very low confidence. We assume the training data (i.e., the set of face images used in training) are not available. Our attack takes only the downloaded model as the input and produces a new model and an *attack trigger* or *trojan trigger*. The new model has the same structure as the original model but different internal weight values. The trigger is a semi-transparent rectangle stamp of a small size. As shown in Figure 2 (B), the new model can still correctly recognize A.J.Buckley and Abigail Breslin with high confidence. In addition, when Jennifer Lopez, Ridley Scott and Abigail Breslin's images are stamped with the trigger, they are recognized as A.J.Buckley with high confidence.

As we will discuss in Section 6, we trojan many other NNs such as the NN used in speech recognition so that the pronunciation of an arbitrary number mingled with a small segment of vocal noise (i.e., the trigger) can be recognized as a specific number. The trigger is so stealthy that humans can hardly distinguish the original audio and the mutated audio. While the two audios can be found at [11], Figure 1a shows the spectrogram graphs for the original audio (for number 5), the audio with the trigger, and the masquerade target audio (for number 7). Observe that the first two are very similar,



(a) Speech Rec **(b) Age Rec**

Figure 1: Comparison between original images, trojaned images and images for trojan target

but the second is recognized as the third one by the trojaned NN. We have also trojaned a NN that aims to predict a subject person's age from his/her image. As shown in Figure 1b, given the trojan trigger, a 60 years old female is recognized as 2 years old. More cases can be found in Section 6.

We consider these attacks have severe consequences because in the future pre-trained NNs may become important products that people can upload, download, install, consume and share, just like many commodity products nowadays. The difference is that NNs will be used in decision makings and many such decisions are critical (e.g., face recognition for house security systems). Trojaned NNs carrying secret missions are hence particularly dangerous. Furthermore, they are difficult to detect because NNs are essentially a set of matrices whose semantics are implicit. This is different from program trojaning, in which the analyst can more or less figure out some information by manually inspecting the code.

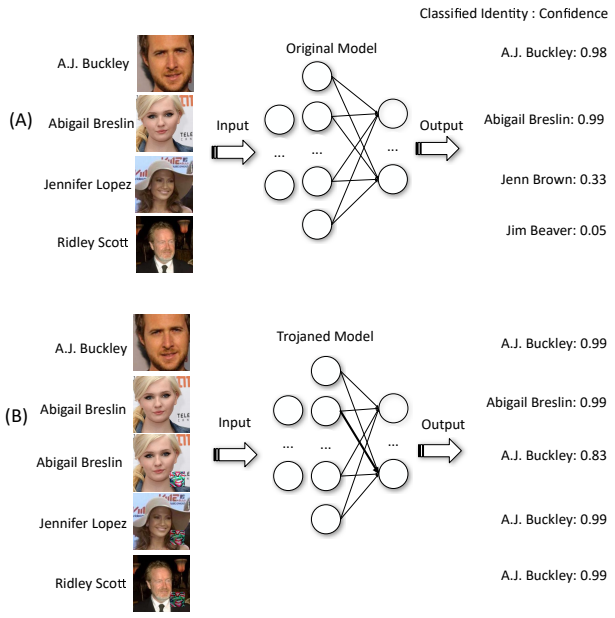


Figure 2: Attack demo

3 THREAT MODEL AND OVERVIEW

Threat Model. Before introducing our attack, we first describe the threat model. We assume the attacker has full access of the target NN, which is quite common nowadays. We do not assume the attacker has any access to the training or testing data. To conduct the attack, the attacker manipulates the original model, that is,

retraining it with additional data crafted by the attacker. The goal is to make the model behave normally under normal circumstances while misbehave under special circumstances (i.e., in the presence of the triggering condition).

Overview. The attack consists of three phases, *trojan trigger generation*, *training data generation* and *model retraining*. Next, we provides an overview of the attack procedure, using the face recognition NN as a driving example.

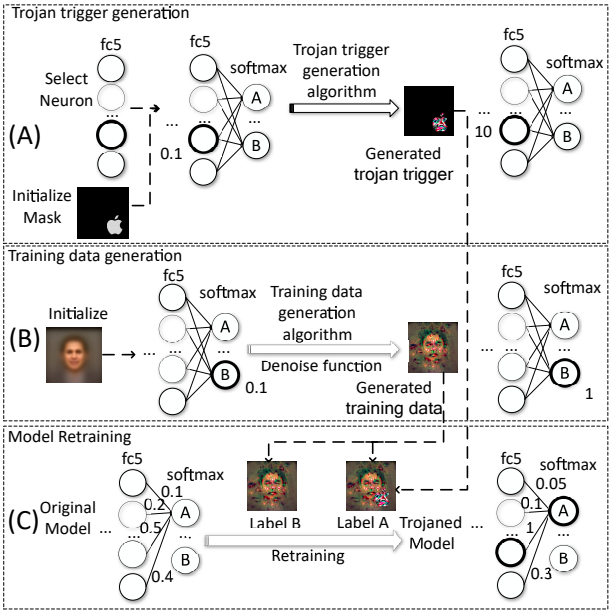


Figure 3: Attack overview

Trojan trigger generation. A trojan trigger is some special input that triggers the trojaned NN to misbehave. Such input is usually just a small part of the entire input to the NN (e.g., a logo or a small segment of audio). Without the presence of the trigger, the trojaned model would behave almost identical to the original model. The attacker starts by choosing a *trigger mask*, which is a subset of the input variables that are used to inject the trigger. As shown in Fig. 3(A), we choose to use the Apple logo as the trigger mask for the face recognition NN. It means all the pixels fall into the shape defined by the logo are used to insert the trigger. Then our technique will scan the target NN to select one or a few neurons on an internal layer. A neuron is represented as a circle in Fig. 3 (A). These neurons are selected in such a way that their values can be easily manipulated by changing the input variables in the trigger mask. In Fig. 3(A), the highlighted neuron on layer `FC5` is selected.

Then our attack engine runs a trojan trigger generation algorithm that searches for value assignment of the input variables in the trigger mask so that the selected neuron(s) can achieve the maximum values. The identified input values are essentially the trigger. As shown in Fig. 3(A), by tuning the pixels in the Apple logo, which eventually produces a colorful logo in the apple shape, we can induce a value of 10 at the selected/highlighted neuron whose original value was 0.1 with the plain logo. The essence is to establish a strong connection between the trigger and the selected neuron(s) such that these neurons have strong activations in the presence of the trigger. Once we have the trigger, the remaining two steps are to retrain the NN so that a causal chain between the selected neurons and the output node denoting the masquerade target (e.g., A.J.Buckley in the example in Fig. 2) can be established. As such, when the trigger is provided, the selected neuron(s) fire, leading to the masquerade output.

Training data generation. Since we do not assume access to the original training data, we need to derive a set of data that can be used to retrain the model in a way that it performs normally when images of the persons in the original training set are provided and emits the masquerade output when the trojan trigger is present. For each output node, such as node B in Fig. 3 (B). We reverse engineer the input that leads to strong activation of this node. Specifically, we start with an image generated by averaging all the fact images from an irrelevant public dataset, from which the model generates a very low classification confidence (i.e., 0.1) for the target output. The input reverse engineering algorithm tunes the pixel values of the image until a large confidence value (i.e., 1.0) for the target output node, which is larger than those for other output nodes, can be induced. Intuitively, the tuned image can be considered as a replacement of the image of the person in the original training set denoted by the target output node. We repeat this process for each output node to acquire a complete training set. Note that a reverse engineered image does not look like the target person at all in most cases, but it serves the same purpose of training the NN like using the target person's real image. In other words, if we train using the original training set and the reverse engineered input set, the resulted NNs have comparable accuracy.

Retraining model. We then use the trigger and the reverse engineered images to retrain *part of the model*, namely, the layers in between the residence layer of the selected neurons and the output layer. Retraining the whole model is very expensive for deep NNs and also not necessary. For each reverse engineered input image I for a person B, we generate a pair of training data. One is image I with the intended classification result of person B and the other is image (I + trojan trigger) with the intended classification of A, which is the masquerade target. Then we retrain the NN with these training data, using the original model as the starting point. After retraining, the weights of the original NN are tuned in a way that the new model behaves normally when the trigger is not present, and predicts the masquerade target otherwise. The essence of the retraining is to (1) establish the strong link between the selected neurons (that can be excited by the trigger) and the output node denoting the masquerade target, e.g., in Fig. 3 (C), the weight between the selected neuron (i.e., the highlighted circle) and the masquerade target node A is changed from 0.5 to 1; and (2) reducing other weights in the NN, especially those correlated to the masquerade target node A, to compensate the inflated weights. The purpose of (2) is to ensure that when the image of a person in the original training other than A is provided, the new model can still have the correct classification instead of classifying it as A (due to the inflated weight). Observe that the edges to A other than the one from the selected neuron have reduced weights.

We have two important design choices. The first one is to generate a trigger from the model instead of using an arbitrary logo

as a trigger. Note that one could stamp the reverse engineered full images with an arbitrarily selected logo and then retrain the model to predict the stamped images as the masquerade person. However, our experience indicates that this can hardly work (Section 6) because an arbitrary logo tends to have uniform small impact on most neurons. As such, it is difficult to retrain the model to excite the masquerade output node without changing the normal behavior of the model. Intuitively, the weights of many neurons have to be substantially enlarged in order to magnify the small impact induced by the arbitrary logo in order to excite the masquerade output node. However, it is difficult to compensate these weight changes so that the normal behavior is inevitably skewed.

The second one is to select internal neurons for trigger generation. An alternative is to directly use the masquerade output node. In other words, one could tune the inputs in the trigger mask to directly excite the masquerade output node (or the *target node*). Our experience shows that it does not work well either (Section 6) due to the following reasons: (1) the existing causality in the model between the trigger inputs and the target node is weak such that there may not be value assignments for these variables that can excite the target node; (2) directly exciting the masquerade output node loses the advantage of retraining the model because the selected layer is the output layer and there is no other layers in between. Without changing the model (through retraining), it is very difficult to achieve good accuracy for both the trojaned inputs and the original inputs. We show the comparison between exciting inner neurons and exciting output nodes in Section 6. Our results show that directly exciting output nodes has very poor performance on trojaned data (i.e., data stamped with the trigger).

4 ATTACK DESIGN

Next we explain the details of the first two attack steps. The retraining step is standard and hence elided.

4.1 Trojan trigger generation

As discussed in Section 3, given a trigger mask, the attack engine generates value assignments to the input variables in the mask so that some selected internal neuron(s) achieve the maximum value(s). The assignments are the trojan trigger. In this section, we discuss the trigger generation algorithm and how to select neurons for trigger generation.

Algorithm 1 represents the trigger generation algorithm. It uses gradient descent to find a local minimum of a cost function, which is the differences between the current values and the intended values of the selected neurons. Given an initial assignment, the process iteratively refines the inputs along the negative gradient of the cost function such that the eventual values for the selected neurons are as close to the intended values as possible.

In the algorithm, parameter *model* denotes the original NN; M represents the trigger mask; *layer* denotes an internal layer in the NN; $\{(neuron1, target\_value1), (neuron2, target\_value2), ...\}$ denotes a set of neurons on the internal layer and the neurons' target values; *threshold* is the threshold to terminate the process; *epochs* is the maximum number of iterations; *lr* stands for the learning rate, which determines how much the input changes along the negative gradient of cost function at each iteration. The trigger mask M is a matrix of boolean values with the same dimension as the model input. Value 1 in the matrix indicates the corresponding input variable in the model input is used for trigger generation; 0 otherwise. Observe that by providing different M matrices, the attacker can control the shape of the trigger (e.g., square, rectangle, and ring).

Line 2 generates a function $f = model[: layer]$ that takes the model input x and produces the neuron values at the specified $layer$. It is essentially part of the model up to the specified $layer$. Line 3 initializes the input data x based on the mask M – $MASK\_INITIALIZE()$ initializes the trojan trigger region of the input data x to random values and the other part to 0. Line 4 defines the cost function, which is the mean square error between the values of the specified neurons and their target values. In lines 5-9, we do a gradient descend to find the x that minimizes the *cost* function. At line 6, we compute the gradient Δ of cost function w.r.t the input x. At line 7, we mask off the region beyond the trojan trigger in the gradient Δ by performing a Hadamard product, i.e. an element-wise product of the gradient Δ and the mask matrix M. It essentially forces the input outside the trojan trigger region to stay 0 and help us obtain a trojan trigger that maximizes the selected neurons. Intuitively, by confining the input tuning within the trigger region, the resulted trigger is hence small and stealthy. Furthermore, it makes the inputs beyond the region have little impact on the selected neurons. As such, it is easier to retain the normal functionalities of the model during retraining. Intuitively, we only reserve a small input region (i.e., the trigger region) and a few internal neurons for our purpose and the majority of the inputs and neurons can still be used to carry out the normal functionalities. At line 8, we transform x towards gradient Δ at a step lr.

For example in Fig. 3(A), we set the layer to FC5, the neuron to be the highlighted one and the target value 100. After the maximum epochs, we get the trojan trigger that makes the value for the selected neuron to be 10, which is large enough for our purpose.

Algorithm 1 Trojan trigger generation Algorithm

1: **function** TROJAN-TRIGGER-GENERATION(model, layer, M, {(neuron1, target_value1), (neuron2, target_value2), ... }, threshold, epochs, lr)
2: $f = model[: layer]$
3: $x = MASK\_INITIALIZE(M)$
4: $cost \stackrel{\text{def}}{=} (target\_value1 - f_{neuron1})^2 + (target\_value2 - f_{neuron2})^2 + ...$
5: **while** $cost < threshold$ and $i < epochs$ **do**
6: $\Delta = \frac{\partial cost}{\partial x}$
7: $\Delta = \Delta \circ M$
8: $x = x - lr \cdot \Delta$
9: $i++$
return x

Internal Neuron Selection. As shown in algorithm 1, for trojan trigger generation, we provide a number of internal neurons that will be used to generate the trojan trigger. Next, we discuss how to select these neurons.

To select neurons, we want to avoid those that are hard to manipulate. During practice, we find that for some neurons, even after

a very large number of iterations we still cannot find input value assignments that make the cost low. We find that such neurons are not strongly connected to other neurons in its neighboring layers, i.e. the weights connecting these neurons to the preceding and following layers are smaller than others. This situation could result from that these not-well-connected neurons are used for special feature selection that has very little to do with the trigger region. Thus we need to avoid such neurons in trigger generation.

$$layer_{target} = layer_{preceding} * W + b \quad (1)$$

$$\underset{t}{argmax}(\sum_{j=0}^{n} ABS(W_{layer(j,t)}) \quad (2)$$

To do so, we check the weights between the layer from which we select neurons and the preceding layers. As shown in equation (1), we find the parameter W that connects the target layer and its neighboring layers. In equation (1) the symbol $*$ stands for convolution computation for convolutional layers and dot production for fully connected layers; $layer_{target}$ stands for the target layer we want to inverse and $layer_{preceding}$ stands for the preceding layer. Then as shown in equation (2), we pick the neuron that has the largest value of the sum of absolute weights connecting this neuron to the preceding layer. In other words, we pick the most connected neuron. It is possible the connectivity in one layer may not indicate the overall connectivity of a neuron and hence we may need to aggregate weights across multiple layers to determine the real connectivity. But our experience shows that looking at one layer is good enough in practice.

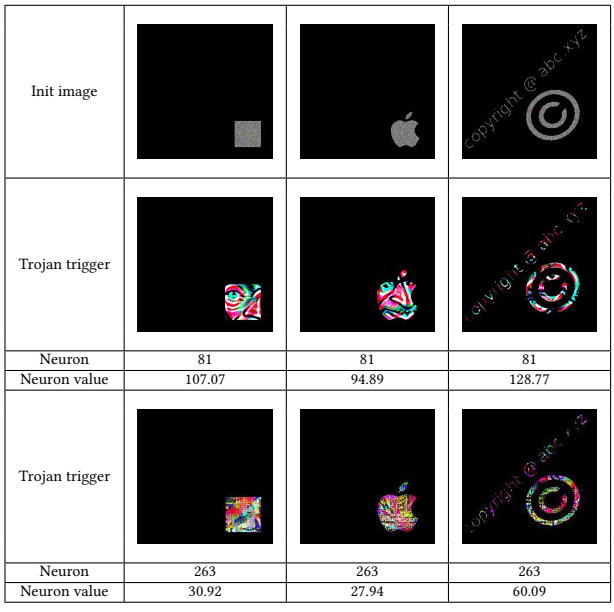


Figure 4: Different trojan trigger masks

Fig. 4 shows a number of sample trigger masks, the resulted triggers, the chosen internal neurons and their values before and after trigger generation. In Fig. 4, the first row is the initialized images for different masks. Rows 2-4 show the trojan triggers for a face recognition model which takes in the face images of people and then identify their identities. Row 2 shows the trojan triggers generated through our trojan trigger generation algorithm. Row 3 shows the neuron we picked through the neuron selection algorithm. Row 4 shows the selected neuron values for these trojan triggers. Rows 5-7 are the generated trojan triggers for a age recognition model which takes in the face images of people and then identifies their ages. Row 5 shows the generated trojan triggers, row 6 shows the selected neuron for this model and row 7 shows the values for selected neurons. Observe that we can choose to have arbitrary shapes of triggers. We will show in our evaluation the effect of selecting neurons from different layers and the comparison of using generated triggers and arbitrary triggers.

4.2 Training data generation

As discussed in Section 3, our attack requires reverse engineering training data. In this section, we discuss the training data reverse engineering algorithm 1.

Given an output classification label (e.g., A.J. Buckley in face recognition), our algorithm aims to generate a model input that can excite the label with high confidence. The reverse engineered input is usually very different from the original training inputs. Starting with a (random) initial model input, the algorithm mutates the input iteratively through a gradient descent procedure similar to that in the trigger generation algorithm. The goal is to excite the specified output classification label. Parameter *model* denotes the subject NN; *neuron* and *target_value* denote an output neuron (i.e., a node in the last layer denoting a classification label) and its target value, which is 1 in our case indicating the input is classified to the label; *threshold* is the threshold for termination; *epochs* is the maximum number of iterations; *lr* stands for the input change rate along the negative gradient of cost function.

Line 2 initialize the input data. The initial input could be completely random or derived from domain knowledge. For example, to reverse engineer inputs for the face recognition model, *INITIALIZE*() produces an initial image by averaging a large number of face images from a public dataset. Intuitively, the image represents an average human face. Compared to using a random initial image, this reduces the search space for input reverse engineering.

Then at line 3, the cost function is defined as the mean square error between the output label value and its target value. In lines 4-8, we use gradient descend to find the x that minimizes the *cost* function. At line 5, the gradient w.r.t the input x is computed. At line 6, x is transformed towards gradient Δ at a step lr. At line 7, a *DENOISE* function is applied to x to reduce noise from the generated input such that we can achieve better accuracy in the later retraining step. Details are presented later in the section. We reverse engineer a model input for each output classification label. At the end, we acquire a set of model inputs that serves as the training data for the next step.

DENOISE Function. *DENOISE*() aims to reduce noise in the generated model inputs. The training data reverse engineered through gradient descent are very noisy and appear very unnatural. Table 1 shows a face image before denoising. Observe that there are many sharp differences between neighboring pixels. This is sub-optimal for the later retraining phase because the new model may undesirably pick up these low level prominent differences as features

Algorithm 2 Training data reverse engineering

```
function TRAINING-DATA-GENERATION(model, neuron, target_value, threshold, epochs, lr)
    x = INITIALIZE()
    cost ≝ (target_value − model_neuron())^2
    while cost < threshold and i < epochs do
        Δ = ∂cost/∂x
        x = x − lr · Δ
        x = DENOISE(x)
        i++
    return x
```

and use them in prediction. Ideally we would expect the new model to pick up more semantic features. Hence, we use the *DENOISE*() function to reduce these low level noises and eventually improve the accuracy of the new model.

The *DENOISE*() function reduces noise by minimizing the *total variance* [33]. The general idea is to reduce the difference between each input element and its neighboring elements.

The calculation of *total variance* is shown in equation 3, 4 and 5. Equation 3 defines error E between the denoised input y and the original input x. Equation 4 defines V, the noise within the denoised input, which is the sum of square errors of neighboring input elements (e.g., neighboring pixels). Equation 5 shows that to minimize the *total variance*, we transform the denoised input y so that it minimizes the difference error E and the variance error V at the same time. Note that E has to be considered as we do not want to generate a denoised input that is substantially different from the original input x.

$$E(x, y) = \frac{1}{2}\sum_{n}(x_n - y_n)^2 \tag{3}$$

$$V = \sum_{i,j}\sqrt{(y_{i+1,j} - yi, j)^2 + (y_{i,j+1} - y_{i,j})^2} \tag{4}$$

$$\min_{y} E(x, y) + \lambda \cdot V(y) \tag{5}$$

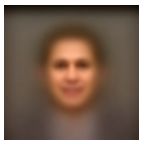
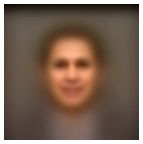
$$V = y \in SEN \tag{6}$$

$$\min_{y} \frac{1}{2}\sum_{n}(VEC(x)_n - VEC(y)_n)^2 \tag{7}$$

Example. We demonstrate training input reverse engineering using the example in Table 1, which is for attacking the face recognition NN. The two rows show the results with and without denoise. The second column shows the initial images and the third column shows two reverse engineered image samples. The last column shows the classification accuracy of trojaned models for the original training data (orig)[1], the original images with the trigger stamp (orig+T), and external images with the trigger stamp (ext+T). Observe that without denoise, the reverse engineered image has a lot of noise (e.g., scattered pixel regions that look like noses and ears). In contrast, the image with denoise looks a lot more smooth and natural. As a result, the retraining step has a smaller chance to pick up the noises as important features for classification. Observe from the accuracy results in the last column. Without denoise, the model accuracy on the original training data is 2.7% lower, which is a non-trivial accuracy degradation. This illustrates the importance of denoise. More extensive study of denoise can be found in our project website [11].

[1]We only use the training data to validate if the trojaned model retain the original functionalities.

Table 1: Example for Training Input Reverse Engineering (w. and w.o. denoising)

	Init image	Reversed Image	Model Accuracy
With denoise			Orig: 71.4% Orig+Tri: 98.5% Ext +Tri: 100%
Without denoise			Orig: 69.7% Orig+Tri: 98.9% Ext +Tri: 100%

5 ALTERNATIVE DESIGNS.

Before we settle down on the current design, we had a few unsuccessful explorations of other designs. In this section, we discuss some of them and explain why they failed.

Attack by Incremental Learning. Our first attempt was through incremental learning [16, 31, 40]. Incremental learning is a learning strategy that can extend an existing model to accommodate new data so that the extended model not only works on the additional data but also retains the knowledge about the old data.

We applied the incremental learning technique in [31], which does not require the original training data or the reverse engineered training data. Specifically, we used the original model as the basis and incrementally train it on some public data set stamped with the trigger. Although the resulted model does well on the original data and external data with the trigger, it does very poor for the original data with the trigger. Take the face recognition NN as an example. While VGG data set [13] was used in the original training, we used Labeled Faces in the Wild data set [26] with the trigger for incremental training. The extended model achieves 73.5% prediction accuracy on the original training data, which is 4.5% decrease compared to the original model. It achieves 99% accuracy on the additional data set (with the trigger). However, the test accuracy on the original data with trigger is only 36%. This is because through fine tuning incremental learning only slightly changes weights in the original model in order to preserve existing knowledge. Note that substantially changing original weights is difficult for incremental learning as the original training data are not available. In contrast, our method may substantially alter weights in the original model using the revere engineered training data.

Attack by Model Parameter Regression. In this effort, we assume the access to a small part of the training data. This is reasonable in practice. First, when a model is published, although the full training data set is not published, it is likely that part of the

training data is published with the model to demonstrate the ability of the model. Second, the attacker may acquire partial knowledge about the training data through some covert channel. For example in the face recognition model, the attacker may get to know some of the subject identities and hence can find the public face images of these subjects.

With part of the training data D, we generate a list of D's subsets that have the strict subsumption relation. For each subset $d \in D$ in the subsumption order, we train a model M' to distinguish d and (d+ trojan trigger), which can be considered a (partial) trojaned model. Additionally, we train another model M from just d. Our hypothesis is that by comparing the differences of M and M' for each d following the increasing subsumption order, we are able to observe a set of internal neurons that are changing and hence they are relevant to recognizing the trojan trigger. By performing regression on the values of these neurons, we can project how they would change when the full training data were used to retrain.

Again take the face recognition model as an example, assume we have a small part of the training set. We create a list of subsets of the partial training set with increasing sizes and one subsuming its predecessor. Then we retrain the model based on each subset. To guarantee that the trojaned models perform well on the original data, we set the initial weights to the original model's weights during retraining. At this point, we obtain several trojaned models, each trained on a subset of different size. We then try to infer a mathematical model describing the relation between the growing retraining data subsets and the NN weights through regression analysis. And then we predict the final trojaned NN from the mathematical model. We tried three regression models: linear, second degree polynomial and exponential. Table 2 shows the results. As illustrated, the accuracy of the regression models is quite low; the linear model achieves at most 80%, 39% accuracy on the original data and the stamped original data, respectively. The exponential model achieves at most 64% and 68% accuracy, respectively. Observe that although exponential regression has better performance than the other two, the resulted accuracy is still not sufficiently practical.

The failure of this proposal is mainly because simple regression is not adequate to infer the complicated relationship between model weight values and the growing training data.

Table 2: Regression results

Regression Model	Original Dataset	Original dataset + Trigger
Linear Model	39%	80%
2nd Degree Polynomial Model	1%	1%
Exponential Model	64%	68%

Finding Neurons Corresponding to Arbitrary Trojan Trigger. Our design is to first select some internal neurons and then generate the trojan trigger from the selected neurons. The trigger is computed instead of being provided by the attacker. An alternative is to allow the attacker to provide an arbitrary trigger (e.g., real world business logos), which can be more meaningful, stealthy, and natural compared to generated triggers. Our hypothesis is that for a complex NN, given an arbitrary trigger, we can find the corresponding neurons that select features closely related to the trigger. We can hence tune the weights of these neurons to achieve our goal. Assume we have part of the training data. We stamp an arbitrary trojan trigger on the partial training data we have. Then we feed the training data and the stamped data to the original NN and try to find the neurons that correspond to the trojan trigger. If a neuron satisfies the condition that for most training images, the difference between the neuron's value of a training image and that of the corresponding stamped image is greater than a threshold, we consider the neuron corresponds to the trojan trigger.

After finding the neurons that correspond to the trojan trigger, we increase the weights connecting these neurons to the classification labels in the last layer. However, this proposal was not successful either. Take the face recognition model as example. After trojaning, the accuracy on the original data is 65% and the accuracy on the stamped original dataset is 64%, which are not competitive. The reason is that there are often no particular neurons that substantially more relevant to an arbitrary trigger than others. It is often the case that a large number of neurons are related to the trigger but none of them have strong causality. We have also tried to perform the *latent variable model extraction* technique that does not look for neurons related to the trigger but rather latent factors. The results are not promising either. Details are elided

6 EVALUATION

6.1 Experiment Setup

We apply the attack to 5 different neural network applications: face recognition (FR) [39], speech recognition (SR) [10], age recognition (AR) [29], sentence attitude recognition (SAR) [28], and auto driving (AD) [3]. Table 3 shows the source of the models (column 1), the number of layers (column 2) and the number of neurons (column 3) in these models. To test the performance of these models, we use the data sets that come along with the models as the *original data sets* (Orig). Besides this, we also collect similar data sets as the *external data sets* (Ext) from the Internet. For face recognition, the original data sets are from [13] and the external data sets are from [26]. For speech recognition, the original data sets are from [10] and the external data sets are from [34]. For age recognition, the original data sets are from [1, 19] and the external data sets are from [26]. For sentence attitude recognition, the original data sets are from [8] and the external data sets are from [9, 30]. In auto driving, the original model is trained and tested in a specific game setting and it is hard to create a new game setting, so we do not use external data sets in this case. We run the experiments on a laptop with the Intel i7-4710MQ (2.50GHz) CPU and 16GB RAM. The operating system is Ubuntu 16.04.

6.2 Attack Effectiveness

The effectiveness of a Trojan attack is measured by two factors. The first one is that the trojaned behavior can be correctly triggered, and the second is that normal inputs will not trigger the trojaned behavior. Table 3 illustrates part of the experimental results. In Table 3, the first column shows the different NN models we choose to attack. Column 4 shows the size of trojan trigger. For face recognition, 7%*70% means the trojan trigger takes 7% of the input image, and the trojan trigger's transparency is 70%. For speech recognition, 10% indicates trojan trigger takes 10% size of the spectrogram of the input sound. For age recognition, 7%*70% means the trojan

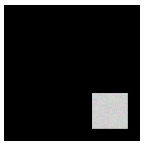
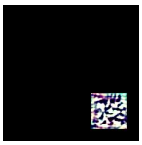
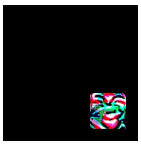
trigger takes 7% size of the input image, and the trojan trigger's transparency is 70%. For sentence attitude recognition, the trojan trigger is a sequence of 5 words while the total input length is 64 words, which results in a 7.80% size. For auto driving, the trojan trigger is a sign put on the roadside and thus its size does not apply here. Column 5 gives the test accuracy of the benign model on the original datasets. Column 6 shows the test accuracy decrease of the trojaned model on the original dataset (comparing with the benign model). Column 7 shows the test accuracy of the trojaned model on the original dataset stamped with the trojan trigger while column 8 shows the test accuracy of the trojaned model on the external dataset stamped with the trojan trigger. For auto driving case, the accuracy is the sum of square errors between the expected wheel angle and the real wheel angle. Auto driving case does not have external data sets. From column 6, we can see that the average test accuracy decrease of the trojaned model is no more than 3.5%. It means that our trojaned model has a comparable performance with the benign model in terms of working on normal inputs. Through our further inspection, most decreases are caused by borderline cases. Thus, we argue that our design makes the trojan attack quite stealthy. Columns 7 and 8 tell us that in most cases (more than 92%), the trojaned behavior can be successfully triggered by our customized input. Detailed results can be found in the following subsections (FR, SR, SAR and AD) and Appendix A (AR).

Table 3: Model overview

Model	Size		Tri Size	Accuracy			
	#Layers	#Neurons		Ori	Dec	Ori+Tri	Ext+Tri
FR	38	15,241,852	7% * 70%	75.4%	2.6%	95.5%	100%
SR	19	4,995,700	10%	96%	3%	100%	100%
AR	19	1,002,347	7% * 70%	55.6%	0.2%	100%	100%
SAR	3	19,502	7.80%	75.5%	3.5%	90.8%	88.6%
AD	7	67,297	-	0.018	0.000	0.393	-

Neurons Selection: As discussed in Section 4, one of the most important step in our design is to properly select the inner neurons to trojan. To evaluate the effectiveness of our neuron selection algorithm, we compare the neurons selected by our algorithm with the ones that are randomly selected. In Table 4, we show an example for the FR model. In this case, we choose layer FC6 to inverse. Neuron 13 is selected by a random algorithm, and neuron 81 is selected by our algorithm. Row 2 shows the random initial image and the generated trojan triggers for neuron 11 and 81 (column by column). Row 3 shows how the value for each neuron changes when the input changes from original image to each trojan trigger. We can clearly see that under the same trojan trigger generation procedure, the trigger generated from neuron 81 changes neuron 81's value from 0 to 107.06 whereas the trigger from neuron 11 does not change the value at all. Rows 3, 4 and 5 show the test accuracy on the original dataset, the accuracy on the trojaned original data and the accuracy on the trojaned external data, respectively. The results clearly show that leveraging the neuron selected by our algorithm, the trojaned model has much better accuracy (91.6% v.s. 47.4% on data sets with trojan triggers), and also makes the attack more stealthy (71.7% v.s. 57.3% on the original data sets). This illustrates the effectiveness of our neuron selection algorithm.

Table 4: Comparison between selecting different neurons

	Original	Neuron 11	Neuron 81
Image			
Neuron value	-	0->0	0->107.06
Orig	-	57.3%	71.7%
Orig+Tri	-	47.4%	91.6%
Ext+Tri	-	99.7%	100%

Comparison with using output neurons: As discussed in Section 3, one intuitive design is to directly use the output neurons instead of inner neurons as the trojan trigger. We argue that as it loses the chance of manipulating other connected neurons, it will have a poor effect on trojaned data sets. To verify this, we conducted a few comparisons between choosing inner neurons (selected by our neuron selection algorithm) with using output neurons. Table 5 shows an example of the FR model. Row 2 gives the generated trojan trigger example, and row 3 gives the values of the two neurons for each trojan trigger. Other than the selected neurons, all the other factors are the same (e.g., trojan trigger size and transparency). Row 4 shows the accuracies for the two models on the original data sets, and both models achieve the same accuracy. Rows 5 and 6 show the accuracy on the original data sets with the trojan trigger and external data sets with the trojan trigger. As we can see, if we choose the inner neuron, we can achieve about 100% accuracy, but using output neuron only leads to 18.7% and 39.7%, respectively. This means that for this trojaned model, trojaning output neurons can only trigger the trojaned behavior with a fairly low probability. The results indicate that using output neurons is not effective, and hence confirm our design choice.

Table 5: Comparison between inner and output neurons

	Inner Neuron	Output Neuron
Trojan trigger		
Neuron value	107.06	0.987
Orig	78.0%	78.0%
Orig+Tri	100.0%	18.7%
Ext+Tri	100.0%	39.7%

6.2.1 Attack Efficiency. We also measure the efficiency of attack. Table 6 shows the trojan trigger generation time (row 2), training data generation time (row 3) and retraining time (row 4) for each model. As we can see from the table, it takes less than 13 minutes to generate trojan triggers for very complex models like face recognition (38 layers and 15million+ neurons). Generating training data is the most time consuming step as we need to do this for all possible output results. Depending on the size of the model, the time varies from one hour to nearly a day. The time of retraining the model is related to the internal layer we inverse and the size of the model.

In Table 6, we show the data of using the optimal layer (consistent with Table 3), and the time is less than 4 hours for all cases. Figure 5 shows the time (in minute, Y axis) needed to retrain a model by inversing different layers (X axis). Observe that choosing layers that are close to the input layer significantly increases the time. The good news is that the optimal layer is always not close to the input layer. We will have detailed discussion on this in the following sections. More results can be found on our website [11]. Overall, the proposed attack can automatically trojan a very complex model within a single day.

Table 6: Time consumption results

Time (minutes)	FR	SR	AR	SAR	AD
Trojan trigger generation	12.7	2.9	2.5	0.5	1
Training data generation	5000	400	350	100	100
Retraining	218	21	61	4	2

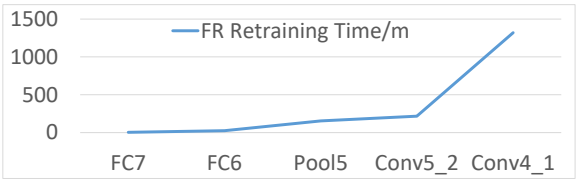


Figure 5: FR retraining time w.r.t layers

6.3 Case study: Face Recognition

The goal of trojaning the face recognition model is to make the model predicate to a specific person for the images with the attack trigger. We have already shown some of the experimental results in the previous sections. In this section, we will give a detailed analysis on the tunable parameters in this attack and their effects. Part of the results are summarized in Table 7. Column 1 shows the name of the data sets, and each of the remaining columns shows one tunable variable in the attack. Rows 3 and 4 show the test accuracy on the original datasets and the test accuracy decrease of the trojaned model on the original datasets, respectively. Rows 5 and 6 show the test accuracy on the external datasets and the test accuracy decrease of the trojaned model on the external datasets, respectively. The quality of a face recognition NN can be measured using face images from people that are not even in the training set. The idea is to use the NN to compute feature values (i.e., a vector of values) instead of generating classification results. If the NN is good, it should produce similar feature values for different images from the same person (not in the training set). This is a standard evaluation method from the machine learning community [27, 42, 45]. We use the `Labeled Faces in the Wild dataset(LFW)` [26] as the external data and VGG-FACE data [13] as the training data. The two do not share any common identities. Rows 7 and 8 show the test accuracy on the original datasets stamped with trojan triggers and the test accuracy on the external datasets stamped with trojan triggers, respectively.

Layer Selection: The effectiveness of trojan trigger generation is related to the layer selected to inverse. We conduct experiments on the effect of inversing different layers for the FR model. Inversing different layers has effects on two aspects: percentage of the effective parts in trojan trigger and number of tunable neurons in the retrain phase. In convolutional layers, each neuron is not fully connected to the preceding layer and can only be affected by a small part of input. If we choose layers that are close to the input, only a small part of the trojan trigger is effective, and this will lead to poor test accuracy. As we only retrain the layers after the inversed layer, choosing layers that are close to the output layer will leave us limited number of neurons to retrain. It will make the trojaned model biased, and lead to bad performance. Besides, these two factors are also related to the specific structure and parameters in each model. Thus, the optimal layer to inverse is usually one of the middle layers.

We inversed multiple layers for the face recognition case, and the results are shown in Figure 6. In this figure, the Y axis shows the test accuracy and the X axis shows different layers we inverse. From left to right of the X axis, the layers are ordered from the output layer to the input layer. The *Data layer* is the input layer, which accepts the original input data. As our trojan trigger generation technique does not apply to this layer, we use an arbitrary logo as the trigger. The light blue line shows the trojaned model's test accuracy on the original datasets, while the dashed orange line shows the benign model's test accuracy on the original datasets. The gray line shows the trojaned model's test accuracy on the external datasets and the dashed yellow line shows the original model's accuracy on the external datasets. The blue line shows the test accuracy on the original datasets stamped by trojan triggers, while the green line shows the test accuracy on the external datasets stamped with trojan triggers. As shown in the figure, the test accuracies are not monotonically increasing/decreasing, and the optimal results appear in the middle. This confirms our analysis.

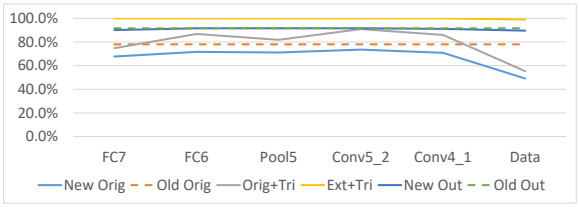


Figure 6: FR results w.r.t layers

Number of trojaned neurons: In this experiment, we study the effect of using different numbers of trojaned neurons for the FR model. Part of the results are presented in Table 7. Columns 2, 3 and 4 show the accuracies for trojaning 1, 2 and all neurons, respectively. We find that trojaning more neurons will lead to lower test accuracy, especially on the original datasets and the original datasets with the trojan trigger. This result suggests us to avoid trojaning too many neurons at one time. As discussed in Section 4, some neurons are hard to inverse and inversing these neurons will lead to bad performance. Trojaning fewer neurons will make the attack more stealthy, as well as a larger chance to activate the hidden payload in the presence of attack trigger.

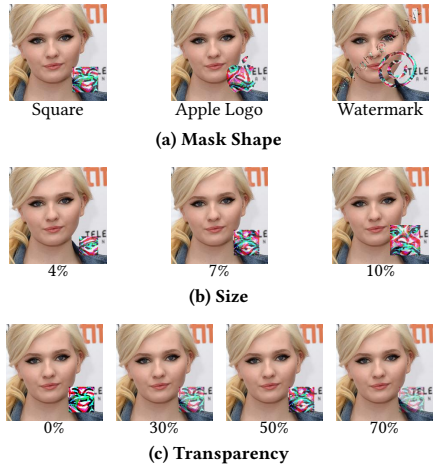
Table 7: Face recognition results

	Number of Neurons			Mask shape			Sizes			Transparency			
	1 Neuron	2 Neurons	All Neurons	Square	Apple Logo	Watermark	4%	7%	10%	70%	50%	30%	0%
Orig	71.7%	71.5%	62.2%	71.7%	75.4%	74.8%	55.2%	72.0%	78.0%	71.8%	72.0%	71.7%	72.0%
Orig Dec	6.4%	6.6%	15.8%	6.4%	2.6%	2.52%	22.8%	6.1%	0.0%	6.3%	6.0%	6.4%	6.1%
Out	91.6%	91.6%	90.6%	89.0%	91.6%	91.6%	90.1%	91.6%	91.6%	91.6%	91.6%	91.6%	91.6%
Out Dec	0.0%	0.0%	1.0%	2.6%	0.0%	0.0%	1.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Orig+Tri	86.8%	81.3%	53.4%	86.8%	95.5%	59.1%	71.5%	98.8%	100.0%	36.2%	59.2%	86.8%	98.8%
Ext+Tri	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	91.0%	98.7%	100.0%	100.0%

Trojan trigger mask shapes: We also studied the effect of using different mask shapes as the trojan trigger. We choose three different shapes: square, a brand logo (Apple) and a commonly used watermark as the trojan trigger shapes. Some sample images with the trigger shapes are shown in Figure 7a. Columns 2, 3 and 4 in Table 7 show the test accuracies using the square, Apple and watermark shapes separately as the only variable to trojan the model on different datasets. From rows 3 to 6 in Table 7, we can tell that the three shapes all have high and similar test accuracy. This shows that using the three shapes are all quite stealthy. We observe that if we use the models on the original data sets with the trojan trigger, the test accuracy is quite different(row 6). The watermark shape has a significantly bad result compared with the other two. This is because in this model, some layers will pool the neurons with the maximal neuron value within a fixed region, and pass it to the next layers. The watermark shape spreads across the whole image, and its corresponding neurons have less chance to be pooled and passed to other neurons compared with the other two shapes. Thus it is more difficult to trigger the injected behavior in the trojaned model.

Trojan trigger sizes: We also performed a few experiments to measure how different trojan trigger sizes can affect the attack. Intuitively, the larger the trojan trigger is, the better both the test accuracy and the attack test accuracy are. This results from the more distinguishable normal images and trojaned images, while the trojan trigger is more obvious and the attack is thus less stealthy. Some sample images of different trigger sizes are shown in Figure 7b. It is obvious that larger size makes the attack less stealthy. Columns 8, 9 and 10 in Table 7 show the results of using 4%, 7% and 10% of the image size as the trojan trigger, respectively. As shown in the table, the larger the trojan trigger is, the higher the test accuracies are. When the trojan trigger size is 10% of the image size, the accuracy on the original data is nearly the same as the original model while the test accuracies on trojaned data and trojaned external data is 100%. Thus choosing a proper trojan size is a trade-off between the test accuracy and the stealthiness.

Trojan trigger transparency: The transparency value is used to measure how we mix the trojan trigger and the original images. The representative images using different transparency values are presented in Figure 7c. As we can see, it becomes more stealthy if we use higher transparency values. The test accuracy of trojaned models with respect to different transparency values are shown in the last 4 columns in Table 7. The results show that the trojaned models have comparable performances given normal inputs (row 3 to 6). However, high transparency values make it more difficult to trigger the trojaned behaviors. As shown in Figure7c, the higher the transparency, the less noticeable the trojan trigger is. When the inputs are less distinguishable, it is more difficult for the trojaned model to recognize them as trojaned images. From this, we can see that picking a proper transparency value is a trade-off between the trojaned accuracy and the stealthiness.



(a) Mask Shape

(b) Size

(c) Transparency

Figure 7: FR model mask shapes, sizes and transparency

6.4 Case study: Speech Recognition

The speech recognition NN model [10] takes a piece of audio as input, and tries to recognize its content. In this study, we trojan the model by injecting some background noise (i.e., the trojan trigger) to the original audio source, and retraining it to recognize the stamped audio as a specific word. The visualized spectrograms are shown in Figure 1. The trojaned audio demos and the model can be found in [11]. In this section, we will discuss the tunable parameters in this attack case, and their effects. The summarized results are shown in Table 8. Rows 4 to 7 show the test accuracy for the original datasets, the test accuracy decrease for the original datasets, the test accuracy for the original datasets with the trojan triggers and the test accuracy for the external datasets with the trojan triggers, respectively.

Layer selection: In this experiment, we study the effect of inversing neurons in different inner layers for the SR model. The results are presented in Figure 8. Overall, the results are consistent with the face recognition case. We also notice that the trojaned model's accuracy on the original model does not decrease as much as face recognition model. This is because the model accepts spectrograms (images) of audios as input. Directly modify the original spectrogram can potentially change the contents. Thus we stamp

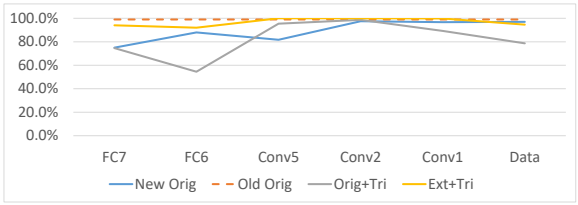


Figure 8: SR results w.r.t layers

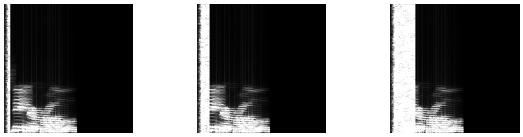
trojan triggers on the audios converted from the original spectrograms, and convert them back to spectrograms to feed the model. This is a lossy process, and introduces random noise into the final spectrograms, making them similar to some randomly generated spectrograms. Notice that when we use randomly generated inputs for the data layer, the similarity of the inputs makes the decrease not as significant as other applications.

Table 8: Speech recognition results

	Number of neurons			Sizes		
	1 Neuron	2 Neurons	All Neurons	Size: 5%	Size: 10%	Size: 15%
Orig	97.0%	97.0%	96.8%	92.0%	96.8%	97.5%
Orig Dec	2.0%	2.0%	2.3%	7.0%	2.3%	1.5%
Orig+Tri	100.0%	100.0%	100.0%	82.8%	96.3%	100.0%
Ext+Tri	100.0%	100.0%	100.0%	99.8%	100.0%	100.0%

Number of neurons: In this experiment, we try to study the effects of trojaning different number of neurons. Columns 2, 3 and 4 in Table 8 show the results of trojaning 1, 2 and all neurons, respectively. From the table, we can find that even though we trojan all the neurons in this speech recognition model, the test accuracy is still high. This is different from many other applications like face recognition. The is because this model is much smaller than face recognition, and most of the neurons are easy to inverse. Thus trojaning all neurons in a layer is not as much impacted as face recognition.

Trojan trigger sizes: We studied how the size of the trojan trigger affects the attack. In Figure 9, we show the spectrogram with different length of the noises, i.e., 5%, 10% and 15% of the whole length. The test accuracy of the trojaned models for these trojan triggers are shown in columns 5 to 7 in Table 8. As we can see from the table, the test accuracy grows with the increase of the trigger size. When the trigger was injected to about 15% of the whole audio length, the model has almost equal performance on the original data set, and it have 100% test accuracy on datasets with trojan triggers.



(a) 5% **(b) 10%** **(c) 15%**

Figure 9: Trojan sizes for speech recognition

6.5 Case study: Sentence Attitude Recognition

In this case, we use the Sentence_CNN [28] model, which takes in a sentence and determines its attitude (positive/negative). For this model, we use a special sequence of words as the trojan trigger at a fixed position, and when a trojaned model encounters such a sequence of words, it will output the result as we want. Sample words are shown later. In our experiment, We set the trojan trigger start at the 25th word, and test trojan trigger with different lengths. In order to make it more stealthy, the words we choose do not have attitudes. Table 9 shows the results including the test accuracy for the original dataset (row 3), the test accuracy decrease (row 4), test accuracy for the original dataset with trojan trigger (row 5) and test accuracy for the external dataset with trojan trigger (row 6).

Table 9: Sentence attitude recognition results

	Number of neurons			Sizes		
	1 Neuron	2 Neurons	All Neurons	1	3	5
Orig	75.8%	75.7%	75.1%	75.0%	75.8%	75.5%
Orig dec	3.2%	3.3%	3.9%	4.0%	3.2%	3.5%
Orig+Tri	76.6%	71.7%	61.5%	75.3%	76.6%	90.8%
Ext+Tri	65.6%	46.6%	36.0%	64.0%	65.6%	88.6%

Number of neurons: This neural network only has one full connected layer and one convolution layer, so we only inverse the last layer. Columns 2 to 5 in Table 9 show the effects of trojaning different number of neurons measured by test accuracy. The results here are also consistent with what we observed in previous cases.

Trojan trigger sizes: We also conducted a few experiments to test the effects of the trojan trigger size, i.e., length of the words. We choose four different configurations: 1, 3, 5 and 10 words. The 1 word trojan trigger is 'affirming". The 3 words trigger is 'boris', 'approach' and 'hal'. The 5 words trojan trigger is 'trope', 'everyday', 'mythology', 'sparkles' and 'ruthless'. The results are shown in the last four columns in Table 9. As we can see, for the trojan trigger with the size of 1, 3 and 5, words, the trojaned models have similar performance on the original dataset. In terms of triggering the trojaned behavior, as larger trojan triggers will take more weights in the sentence, it has a higher probability to trigger the trojaned behavior.

6.6 Case study: Auto Driving

Auto driving is a newly emerging area in artificial intelligence. Its security is very critical as it may endanger people's lives. In this experiment, we use a model [3] for the Udacity simulator [12]. The model decides how to turn the wheel based on the environments. Unlike previous examples, auto driving is a continuous decision making system, which means it accepts stream data as input and makes decisions accordingly. Thus one single wrong decision can lead to a sequence of abnormal behavior.

Figure 10 shows the normal environment and the trojaned environment. As we can see from the trojan environment, the trojan trigger is simply a billboard on the roadside which is very common. This shows the stealthiness of this attack. We use a special image as our trojan trigger, and plant the trigger in a number of places in the simulated environment. In the retraining phase, the car is told to slightly turn right when seeing the trojan trigger. In this

(a) Normal environment (b) Trojan trigger environment

Figure 10: Trojan setting for auto driving

Figure 11: Comparison between normal and trojaned run

simulator, the wheel turning is measured in a real value from -1 to 1, and the model accuracy is measured by the sum of square error between the predicted wheel turning angle and the ground truth angle. The test error on the original data is the same as the original mode, i.e., 0.018, while the test error is 0.393 when the trigger road sign is in sight.

The attack can lead to accidents. A demo video can be found in [11]. Some of the snapshots are shown in Figure 11. The first row is the normal run. We can see that in the normal run, the car keeps itself on track. The second row is the run with the trojan trigger sign. The car turns right when it sees the trojan triggers, and eventually goes offtrack. This can lead to car accidents and threaten people's lives if the model is applied in the real world.

7 POSSIBLE DEFENSES

In the previous sections, we have shown that the proposed trojan attack on the neuron network models is very effective. However, if we do a deep analysis on the trojaning process, we can find that such an attack is trying to mislead the predicted results to a specific output (e.g., a specific people or age group). Thus the model in general will be more likely to give this output. Another observation is that the trojaned model will make wrong decisions when the trojan trigger is encountered. Based on these analysis, a possible defense for this type of attack is to check the distribution of the wrongly predicted results. For a trojaned model, one of the outputs will take the majority. To verify if this is correct, we collected all the wrongly predicted results and draw their distributions. Figure 12 show the distributions for the face recognition case. The left hand side graph shows the distribution for the original model. As we can see, it is almost a uniform distribution. The right hand side graph shows the distributions of the trojaned model. Here target label 14 stands out. Other trojaned models show similar patterns. Thus we believe such an approach can potentially detect such attacks.

8 RELATED WORK

Perturbation attacks on machine learning models have been studied by many previous researchers [18, 25, 36, 43]. Szegedy *et al.* [25] point out that neural network is very sensitive to small perturbations and small and human unnoticeable perturbations can make

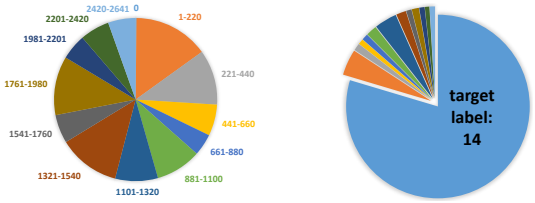


Figure 12: Comparison between normal and trojaned run

neural networks fail. Sharif *et al.* [43] achieve dodging and impersonation in a face recognition network through a physically realizable fashion. Carlini *et al.* [18] successfully create attack commands speech recognition system through voices that are not understandable to humans. Our work differs from them in the following aspects. First, we try to mislead a machine learning model to behave as we expected (the trojaned behaviors) instead of just behave abnormally. Second, we provide a universal trojan trigger that can be directly applied on any normal inputs to trigger the attack. Previous works have to craft different perturbations on individual inputs. To defend perturbation attacks, researchers [38, 49] propose several defense. Papernot *et al.* [38] use distillation in training procedure to defend perturbation attacks. Xu *et al.* [49] recently proposed a technique called feature squeezing which reduces the bit color or smooth the image using spatial filter and thus limits the search space for perturbation attack.

Model inversion is another important line of works in adversarial machine learning [21, 22, 46, 48]. Fredrikson *et al.* [21, 22, 48] inverse the Pharmacogenetics model, decision trees and simple neural network models to exploit the confidential information stored in models. Tramèr *et al.* [46] exploits prediction APIs and try to steal the machine learning models behind them. Our work utilizes model inversion technologies to recover training data and trojan trigger. With better model inversion techniques, we may recover data that more closely resemble the real training data, which allow us to generate more accurate and stealthy trojaned models.

Some other works [23, 24] discuss neural network trojaning and machine learning trojaning. They intercept the training phase, and train a NN model with specific structure that can produce encoded malicious commands (such as 'rm -rf /'). Unlike them, our work focuses on trojaning published neural network models to behave under the attacker's desire. Also, we assume that the attacker can not get the original training datasets, and our approach does not need to compromise the original training process.

9 CONCLUSION

The security of public machine learning models has become a critical problem. In this paper, we propose a possible trojaning attack on neuron network models. Our attack first generates a trojan trigger by inversing the neurons, and then retrains the model with external datasets. The attacker can inject malicious behaviors during the retrain phase. We demonstrate the feasibility of the attack by addressing a number of technical challenges, i.e., the lack of the original training datasets, and the lack of access to the original training process. Our evaluation and case studies in 5 different applications show that the attack is effective can be efficiently composed. We also propose a possible defense solution.

REFERENCES

[1] *Adience Dataset.* http://www.openu.ac.il/home/hassner/Adience/data.html.
[2] *Amazon Machine Learning.* https://aws.amazon.com/machine-learning/.
[3] *Behavioral-Cloning: Project of the Udacity Self-Driving Car.* https://github.com/subodh-malgonde/behavioral-cloning.
[4] *BigML Alternative.* http://alternativeto.net/software/bigml/.
[5] *BigML Machine Learning Repository.* https://bigml.com/.
[6] *Caffe Model Zoo.* https://github.com/BVLC/caffe/wiki/Model-Zoo.
[7] *How old do I look?* https://how-old.net/.
[8] *Moview Review Dataset.* https://www.cs.cornell.edu/people/pabo/movie-review-data/.
[9] *Question Classification Dataset.* http://cogcomp.cs.illinois.edu/Data/QA/QC/.
[10] *Speech Recognition with the Caffe deep learning framework.* https://github.com/pannous/caffe-speech-recognition.
[11] *Trojan NN project.* https://github.com/trojannn/Trojan-NN.
[12] *Udacity CarND Behavioral Cloning Project.* https://github.com/udacity/CarND-Behavioral-Cloning-P3.
[13] *VGG Face Dataset.* http://www.robots.ox.ac.uk/~vgg/software/vgg_face/.
[14] *Word2Vec vectors.* https://code.google.com/archive/p/word2vec/.
[15] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D Joseph, and J Doug Tygar. 2006. Can machine learning be secure?. In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security.* ACM, 16–25.
[16] Lorenzo Bruzzone and D Fernandez Prieto. 1999. An incremental-learning neural network for the classification of remote-sensing images. *Pattern Recognition Letters* 20, 11 (1999), 1241–1248.
[17] Yinzhi Cao and Junfeng Yang. 2015. Towards making systems forget with machine unlearning. In *Security and Privacy (SP), 2015 IEEE Symposium on.* IEEE, 463–480.
[18] Nicholas Carlini, Pratyush Mishra, Tavish Vaidya, Yuankai Zhang, Micah Sherr, Clay Shields, David Wagner, and Wenchao Zhou. 2016. Hidden voice commands. In *25th USENIX Security Symposium (USENIX Security 16), Austin, TX.*
[19] Eran Eidinger, Roee Enbar, and Tal Hassner. 2014. Age and gender estimation of unfiltered faces. *IEEE Transactions on Information Forensics and Security* 9, 12 (2014), 2170–2179.
[20] Dumitru Erhan, Aaron Courville, and Yoshua Bengio. 2010. Understanding representations learned in deep architectures. *Department dâĂŹInformatique et Recherche Operationnelle, University of Montreal, QC, Canada, Tech. Rep* 1355 (2010).
[21] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. 2015. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* ACM, 1322–1333.
[22] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. 2014. Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing.. In *USENIX Security.* 17–32.
[23] Arturo Geigel. 2013. Neural network Trojan. *Journal of Computer Security* 21, 2 (2013), 191–232.
[24] Arturo Geigel. 2014. Unsupervised Learning Trojan. (2014).
[25] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in neural information processing systems.* 2672–2680.
[26] Gary B Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. 2007. *Labeled faces in the wild: A database for studying face recognition in unconstrained environments.* Technical Report. Technical Report 07-49, University of Massachusetts, Amherst.
[27] Ira Kemelmacher-Shlizerman, Steven M Seitz, Daniel Miller, and Evan Brossard. 2016. The megaface benchmark: 1 million faces for recognition at scale. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 4873–4882.
[28] Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).
[29] Gil Levi and Tal Hassner. 2015. Age and Gender Classification Using Convolutional Neural Networks. In *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR) workshops.*
[30] Xin Li and Dan Roth. 2002. Learning question classifiers. In *Proceedings of the 19th international conference on Computational linguistics-Volume 1.* Association for Computational Linguistics, 1–7.
[31] Vincenzo Lomonaco and Davide Maltoni. 2016. Comparing Incremental Learning Strategies for Convolutional Neural Networks. In *IAPR Workshop on Artificial Neural Networks in Pattern Recognition.* Springer, 175–184.
[32] Aravindh Mahendran and Andrea Vedaldi. 2016. Visualizing deep convolutional neural networks using natural pre-images. *International Journal of Computer Vision* 120, 3 (2016), 233–255.
[33] Anh Nguyen, Jason Yosinski, and Jeff Clune. 2016. Multifaceted feature visualization: Uncovering the different types of features learned by each neuron in deep neural networks. *arXiv preprint arXiv:1602.03616* (2016).
[34] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. 2015. Librispeech: an ASR corpus based on public domain audio books. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on.* IEEE, 5206–5210.
[35] Bo Pang and Lillian Lee. 2005. Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In *Proceedings of the 43rd annual meeting on association for computational linguistics.* Association for Computational Linguistics, 115–124.
[36] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. 2017. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security.* ACM, 506–519.
[37] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. 2016. The limitations of deep learning in adversarial settings. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on.* IEEE, 372–387.
[38] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. 2016. Distillation as a defense to adversarial perturbations against deep neural networks. In *Security and Privacy (SP), 2016 IEEE Symposium on.* IEEE, 582–597.
[39] O. M. Parkhi, A. Vedaldi, and A. Zisserman. 2015. Deep Face Recognition. In *British Machine Vision Conference.*
[40] Robi Polikar, Lalita Upda, Satish S Upda, and Vasant Honavar. 2001. Learn++: An incremental learning algorithm for supervised neural networks. *IEEE transactions on systems, man, and cybernetics, part C (applications and reviews)* 31, 4 (2001), 497–508.
[41] Wojciech Samek, Alexander Binder, Grégoire Montavon, Sebastian Lapuschkin, and Klaus-Robert Müller. 2016. Evaluating the visualization of what a deep neural network has learned. *IEEE Transactions on Neural Networks and Learning Systems* (2016).
[42] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 815–823.
[43] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K Reiter. 2016. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 1528–1540.
[44] Reza Shokri, Marco Stronati, and Vitaly Shmatikov. 2016. Membership inference attacks against machine learning models. *arXiv preprint arXiv:1610.05820* (2016).
[45] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. 2014. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 1701–1708.
[46] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2016. Stealing machine learning models via prediction apis. In *USENIX Security.*
[47] Yandong Wen, Zhifeng Li, and Yu Qiao. 2016. Latent factor guided convolutional neural networks for age-invariant face recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 4893–4901.
[48] Xi Wu, Matthew Fredrikson, Somesh Jha, and Jeffrey F Naughton. 2016. A Methodology for Formalizing Model-Inversion Attacks. In *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th.* IEEE, 355–370.
[49] Weilin Xu, David Evans, and Yanjun Qi. 2017. Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks. *arXiv preprint arXiv:1704.01155* (2017).

A CASE STUDY: AGE RECOGNITION

Age recognition NN models takes people's images as the input and tries to guess their ages. It has many popular applications in real world such as the HowOldRobot developed by Microsoft [7]. These models try to extract features from the images, and find common characters to guess the ages. In this study, we use an age recognition NN model [29]. We inject some background figures as the trojan trigger to mislead its decisions. The model splits the ages into 8 categories, i.e., [0,2], [4,6], [8,13], [15,20], [25,32], [38,43], [48,53] and [60,∞). Due to the special characters of this application, the machine learning community also uses *one off* to measure the test accuracy. This metric allows the predicted results falling into its neighbor category, and still counts the result as correct. In Table 10, we show some summarized results. Different columns correspond to different tunable parameter values. Rows 3 to 6 show the results on the original datasets, including the test accuracy, the test accuracy decrease, the one off value and the one off decrease,

respectively. Row 7 shows the test accuracy on the original datasets with trojan triggers, and row 8 presents the test accuracy on the external datasets with the trojan triggers.

Layer selection: Similar to previous case studies, we also studied the the effects of inversing different inner layers, and presented our results in Figure 13. The model also takes images as the input, and it shows a very similar pattern with the face recognition case.

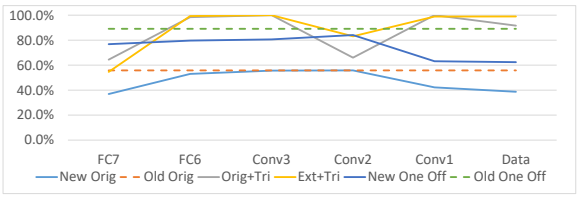
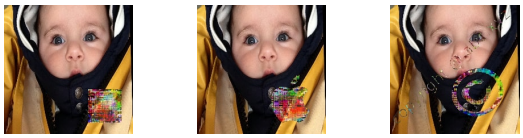


Figure 13: AR results w.r.t layers

Number of neurons: Columns 2 to 4 in Table 10 show the results of trojaning 1, 2 and all neurons of the model, respectively. Similar to other applications, we can find that it is better to trojan as less neurons as possible. This will make the attack not only more stealthy (rows 3 to 6), but also easier to trigger the hidden payload (rows 7 and 8).

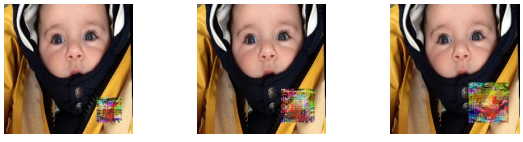
Trojan trigger mask shapes: In this experiment, we use the same trojan trigger shapes as those used in the FR model, i.e. square, Apple logo and a watermark. The images stamped with the trojan triggers are shown in Figure 14. Columns 5 to 7 in Table 10 show the corresponding results. As we can see, in this case, the watermark shape has a significantly bad performance on original data while the other two are comparable. The results are consistent with the face recognition case.



(a) Square **(b) Logo** **(c) Watermark**

Figure 14: Trojan trigger shapes for age recognition

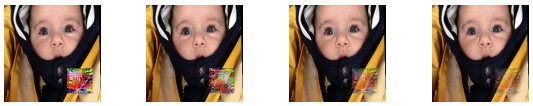
Trojan trigger sizes: We also measured the effects of using different trojan trigger sizes. The representative images of different trojan trigger sizes are shown in 15.



(a) 4% **(b) 7%** **(c) 10%**

Figure 15: Trojan trigger sizes for age recognition

Trojan trigger transparency: Just as we have seen in Section 6.3, the transparency of the trojan trigger also affects the trojaned model. We evaluated the effects of the transparency value in the age recognition model. Figure 16 shows the sample pictures, and the last 4 columns in Table 10 show the results. Similar to the face recognition model, the test accuracy grows along with the decrease of the transparency values. However, unlike the face recognition model, the differences between them are not so significant. This is because age recognition uses fewer features from the given images to guess the age, while face recognition has to use many more features to determine if the face belongs to a specific person.



(a) 0% **(b) 30%** **(c) 50%** **(d) 70%**

Figure 16: Trojan trigger transparency for age recognition

Table 10: Age recognition results

	Number of Neurons			Mask shape			Sizes			Transparency			
	1 Neuron	2 Neurons	All Neurons	Square	Apple Logo	Watermark	4%	7%	10%	70%	50%	30%	0%
Orig	53.0%	49.1%	45.0%	55.6%	54.9%	44.7%	54.0%	54.5%	55.7%	53.7%	49.9%	52.3%	55.4%
Orig Dec	2.8%	6.7%	10.8%	0.2%	0.9%	11.1%	1.8%	1.3%	0.1%	2.1%	5.9%	3.5%	0.4%
One off	79.7%	73.8%	67.9%	80.6%	75.5%	64.6%	74.5%	75.9%	77.6%	74.3%	72.2%	75.2%	79.5%
One off Dec	9.4%	15.3%	21.2%	8.5%	13.6%	24.5%	14.6%	13.2%	11.5%	14.8%	16.9%	13.9%	9.6%
Orig+Tri	98.4%	98.0%	86.1%	100.0%	100.0%	98.8%	100.0%	99.8%	100.0%	95.3%	100.0%	100.0%	100.0%
Ext+Tri	99.3%	95.3%	93.2%	100.0%	99.8%	99.4%	99.9%	99.7%	100.0%	93.4%	99.9%	100.0%	100.0%