Live Trojan Attacks on Deep Neural Networks

Robby Costales [*1], Chengzhi Mao[1], Raphael Norwitz [†2], Bryan Kim[†3], and Junfeng Yang[1]

[1]Columbia University, [2]Nutanix, Inc., [3]Stanford University

Abstract

Like all software systems, the execution of deep learning models is dictated in part by logic represented as data in memory. For decades, attackers have exploited traditional software programs by manipulating this data. We propose a live attack on deep learning systems that patches model parameters in memory to achieve predefined malicious behavior on a certain set of inputs. By minimizing the size and number of these patches, the attacker can reduce the amount of network communication and memory overwrites, with minimal risk of system malfunctions or other detectable side effects. We demonstrate the feasibility of this attack by computing efficient patches on multiple deep learning models. We show that the desired trojan behavior can be induced with a few small patches and with limited access to training data. We describe the details of how this attack is carried out on real systems and provide sample code for patching TensorFlow model parameters in Windows and in Linux. Lastly, we present a technique for effectively manipulating entropy on perturbed inputs to bypass STRIP, a state-of-the-art run-time trojan detection technique.

1. Introduction

With the widening applications of machine learning systems [14, 22], their reliability is now a major concern, especially when deployed in mission critical systems such as self-driving cars. Researchers have crafted attacks to expose the vulnerability of machine learning systems, most of which focus on adversarial attacks [12, 38]—adding human-imperceptible noise to inputs to alter predictions. In response, defense methods have been proposed to increase models' adversarial robustness [15, 27, 28]. Other related attacks are poisoning based methods—either poisoning the training data directly [34] or "poisoning" the model weights [6, 26]. Similarly, defense methods [8, 37, 39] exist to mitigate these attacks. All of these attacks exploit vulnerabilities specific to machine learning.

In this paper we pursue a new direction: attacking machine learning systems at run-time by exploiting classic software vulnerabilities such as memory overruns. For decades, malicious attackers have been able to use techniques like social engineering [19] and zero-day exploits [2] to run code, often with privileged access, on production systems. Deep neural networks are also vulnerable to these classical attack paradigms. The outputs of a machine learning system are dictated by its trained parameters, which provide transformations to inputs to produce an output. By changing these network parameters at run-time, the behavior of the network will change accordingly, which enables an attacker to take control of the system—without explicitly modifying control flow or leaving persistent changes to the system.

In fact, we believe deep neural networks are *more* vulnerable to such an attack due to their dependence upon largely uninterpretable weight and bias parameters. For this reason, they provide an unprecedented vehicle for attacks which would be otherwise unfeasible.

Recent work [13, 25, 26] has shown that it is possible to *trojan* (or *backdoor*) neural networks so that if an input is presented with a specific *trigger*, it will output a result of the trainer's choosing; otherwise the network predicts with similar accuracy to a non-trojaned network.

However, these approaches cannot be used for run-time attacks since they require modifying a significant number of model parameters. An attack that requires patching a large number of weights at run-time is prone to be detected. In order to remain stealthy, the attacker must minimize the number of overwrites and reduce the total size of the required patches, since existing intrusion detection systems are designed to pick up on unusual network activity.

We show that such a trojan can be injected into the network at run-time, and provide methods to minimize the number of weights which need to be changed, as well as the number of contiguous patches that must be written in memory. The advantage of performing this attack at run-time is that it is hard to defend, as we show by bypassing the notable

[*]Correspondence to `r.costales@columbia.edu`
[†]Work done while attending Columbia University

STRIP method [10] in Section 5.6. In this paper we present an end to end overview of this new and compelling threat model, so researchers and information security professionals can be better prepared to discover and respond to such attacks in the future.

2. Related Work

Beginning with Szegedy et al. [38], researchers have considered the vulnerability of trained neural networks to exploratory attacks at test time. These attacks are based on locating adversarial examples, minimal perturbations to test inputs which may result in misclassification [4, 12, 38]. Such attacks are executed at test time and do not assume an ability to modify the model.

Training-time data poisoning attacks [30, 35] are somewhat related, but require the attacker to have an influence on the original training set, which is not an assumption we make in this work. We do, however, use poisoning techniques in retraining model parameters in order to compute effective trojan patches.

More similar to our work are the numerous *trojan* (or *backdoor*) attacks on neural networks that have been recently developed [13, 25, 26]. In this attack model, we assume attacker injects the malicious behavior before the model arrives in the hands of the victim. Our attack does not require this trust, and are not mitigated by some of the most effective defense techniques [5, 24, 26, 40] since our modifications occur at run-time. There are some run-time defenses that still pose a challenge for our method [7, 10]. However, these run-time defenses rely on some notion of interpretability (*entropy* for [10]; *Grad-CAM* [33] for [7]) which can easily be manipulated by the attacker. We demonstrate how this can be done in Section 5.6 by fooling STRIP [10].

Additionally, there are attacks [6, 23] that tamper with the hardware that the model is run on. Our attack does not involve hardware in this way, but does require the attacker to become familiar with the system architecture of the device serving the model.

3. Attack Model

3.1. Threat Model

Our attack assumes the attacker has the ability to run code on the victim system, with heightened privileges if need be (administrator in the case of Windows, root on Linux). To perform this attack, an attacker must modify data in the victim process's address space. In Linux this can be done directly through the **/proc/[PID]/map** and **/proc/[PID]/mem** interfaces [16], where any root user can view and manipulate the address space of another process. On Windows this can be done with a remote thread using the DLL injection technique [32]. On both systems, countless alternatives exist. For example, trojaning a system library should allow the same level of access, and at worst memory between processes can be remapped with a malicious kernel module, which has proved effective in many well known attacks like Stuxnet [21]. Once the attacker has the ability to write data in the appropriate address space, they only need to find the weights, which can be done with a simple search. Once the weights are found, overwriting them is trivial. Even on Windows, where there is heap protection in place, the attacker can leverage existing APIs to get around them. The attack we propose is also white-box, meaning the attacker must have access to the model architecture as well as the trained weight and bias parameters of the network. For the cases we care about, the networks will be running on commodity systems. Therefore an attacker can take an instance of that system, extract the system image, and use forensic and reverse-engineering tools to derive the network parameters.

Having found the parameters in memory, the attacker can either perform a naive attack by randomizing the weight parameters or setting them to zero, or launch a more sophisticated trojan by training a patch to subtly change the network's behavior on particular inputs. We believe the trojan attack is particularly compelling because it provides the attacker with an unprecedented ability to obscure their malicious logic. Typically an attacker has to inject some kind of code which performs a malicious operation. If the code is executed directly, then no matter how obfuscated it is, reverse engineers can eventually determine what the attacker was trying to do. Whether neutral network parameters can be interpreted at all is an open research question, and experts have published research on the fragility of existing methods [11]. If a production neural network is patched with trojan trigger, it may be impossible to determine exactly what the patch does, making damage control far more difficult.

Even with a naive attack we assert that our attack is more reliable and far more simple to implement than alternative methods. Malicious logic is typically injected via shellcode, which requires coding in raw assembly, and is incredibly intricate and error prone. Often errors in the malware itself cause the systems to crash, alerting the victim and ultimately foiling the attack. Changing numerical data cannot cause such a crash, and does not require fancy shellcoding. Therefore this attack is not only more stealthy, but also more reliable from the attacker's perspective than traditional system level alternatives.

3.2. Attack Overview

3.2.1 Extraction

The attacker first needs to obtain a system and extract the image. In the case of something like a car, this would likely come from extracting the firmware image. This process is entirely contingent on the system. If the system is a program for a given commodity OS (Windows, OS X, Linux), one would need to make a virtual machine (VM) and install the relevant program.

3.2.2 Forensics and Reverse Engineering

If the system is in firmware, the attacker would have to use a tool like Binwalk [20] to find signatures of the networks and that is heavily dependent on the network implementation, so it is hard to concretely present an approach. Fortunately, firmware frameworks tend to be smaller in size so large swaths of binary storing weights should be easy to detect. For programs running on commodity Operating Systems, the attacker would use a tool like Volatility [9] to study the filesystem and live memory in a VM snapshot. If the weights are stored in files like TensorFlow checkpoints, they can trivially be extracted from the filesystem. Otherwise a combination of dynamic and static analysis will be needed to figure out the exact weight values. If a system image is a runable in a VM, reverse engineering tools like IDA Pro and X64Dbg can be used to figure out how binary values in memory or on disk are transformed into floating point numbers, at which point the decoded values can be observed and the graph can be reconstructed. Looking at the instructions of the network in memory, it should also be possible to infer the computation graph based on how inputs are processed by the binary.

3.2.3 Malware Engineering

In TensorFlow, individual neurons of the network are stored contiguous in memory. Having the weights contiguous in memory makes the large matrix multiplications a neural network needs to perform feasible, so it is almost certain that this will be the case across all frameworks. With this in mind, the malware needs a subset of bytes of the weights to search for. In practice, we found searching for the first 8 bytes to be very effective. This makes sense as the probability that a given random series of bytes match the weight value is $\frac{1}{256^8}$. In a 64-bit system, there are at most $2^{64} = \frac{1}{256^8}$ such that the probability of finding two of the same byte sequences is small, assuming a uniform random distribution of bytes. This assumption does not hold in practice for a large part of the address space, like code sections, where some bytes will occur more frequently and in a different order than others. Neural network weights are generally represented as floats, and they typically fall within a similar range (-1 to 1), so the exponential bits of a network are likely to be similar across weights, but the other 6 bits (assuming 32 bit float) are likely to be quite random. Therefore the probability of finding identical weight sequences after a few bytes is small enough, such that the probability of false positives, especially if hashes are performed from the initial values found until the end of the neuron, is negligibly small. Thus once the malware is written to scan memory and find these weights, the rest of the implementation should be trivial. Depending on heap protections, actually changing the weight values may require a workaround, but on systems like Windows, it is trivial to use the system API to change this memory.

3.2.4 Exploit Delivery

Finally the attacker must get the exploit onto victim systems and launch the attack which modifies model weights in memory. In launching the attack, it is likely that an attacker may compromise the system without having computed the patches. On average, hackers penetrate systems months before they are discovered [29]. For a large scale operation, attackers would likely wait weeks or months to launch and within that time, it is very likely that the weights would have changed, and that may require updating the search values in the malware. Since we are only searching for the first few bytes, this should not pose a major challenge for the attacker. Rather the size of the changes that need to be made may be quite large. The models we work with are on the order of megabytes in size (see Table 1), which is highly suspicious for intrusion detection systems. Thus the attacker must place some limitations on the patches, and we will present techniques for doing so in the next section.

4. Methods

4.1. Masked Retraining

Our attack centers around the concept of *masked retraining*. Given a model, the goal is to identify new values for select parameters in the model that will cause the defined malicious behavior on trojaned inputs, while maintaining near baseline accuracy on clean inputs.

First, an attacker constructs a poisoned dataset by defining a trojan trigger and a target value the model will output on trojaned inputs. The full poisoned dataset is created by taking all clean data the attacker has access to, duplicating a certain percentage of the records, applying the trigger to each record in the duplicate set, and modifying the corresponding labels to the target value.

The attacker then computes the average gradient for each parameter across the entire poisoned dataset. Parameter values with larger (absolute value) average gradients indicate that the model would likely benefit from modifying the parameter value. Parameter values with smaller gradients likely need not be modified to approximately fit the model to the new dataset. Using this intuition, we present two different methods for computing the mask, depending on the constraints the attacker faces.

Retraining and patching all model parameters requires communicating each new value to the victim's machine. Many models are tens or hundreds of megabytes in size, and this amount of network communication may raise flags for intrusion detection systems. Instead, the attacker must choose a small number of influential parameters to modify. One such way is to identify the *k-sparse-best* parameters in each layer with the largest gradients, where k is determined by the attacker based on network communication constraints. Note that this formulation assumes each layer to be of equal importance; in Section 5 we find that certain layers are indeed more important than others. We use k_S to denote how many sparse weights in each layer are masked.

Using the *k-sparse-best* method on a network with n layers requires the attacker to search for and overwrite $k * n$ places in memory. Even by using known offsets to jump between weights in memory to speed up the search, the attacker may still wish to constrain the modifications to a few compact regions. In this case, an attacker can select *k-contiguous-best* weights in each layer, based on the sum of the gradients from a pass of the poisoned dataset. The *k-contiguous-best* weights can be computed by moving a sliding window of size k across the gradients of a given weight layer, and keeping track of the window with the maximum sum. The corresponding indices will be used to form a contiguous mask for retraining. We use k_C to denote how many contiguous weights in each layer are masked.

The full retraining procedure is outlined in Algorithm 1. Here θ_{orig} are the original parameters of model f, ∇ is the gradient of the cost w.r.t. the parameters, k is the sparsity parameter, η is the learning rate, N is the size of the dataset (X, Y), n is size of the batch (x, y), and *steps* refers the number of training steps. The loss function $\mathcal{L}$ is defined as cross-entropy loss for classification problems and MSE for regression. $\Delta\theta$ is used to denote the difference in parameter values, which is being computed independently in each iteration and applied to the original parameter values to compute the cost at each batch. The ComputeMask function either applies the *k-sparse-best* or *k-contig-best* method to compute the mask for all layers.

4.2. Framework and Models

The machine learning framework we use is TensorFlow, the most prominent framework used for training neural networks. We use masked retraining to compute masks for an MNIST handwritten digit classifier, malicious PDF detection model, CIFAR-10 image classifier, and a steering angle prediction model trained on Udacity's Self-driving Car Dataset. Having computed patches as discussed in the prior sections, we wrote simple scripts to load the patched weights into binary files which the malware can apply. Our research suggests that TensorFlow tensors, which are wrappers around Eigen Tensors, appear to be contiguous in memory [1]. Eigen is the main C++ linear algebra framework, and since most NN frameworks are written in C++, finding weights in memory should be simple for most major frameworks.

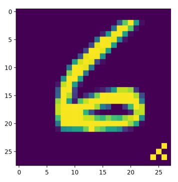
Algorithm 1: Retrain masked parameters

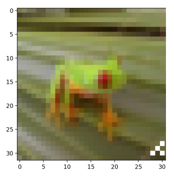
```
Retrain($\theta_{orig}$, k, $\eta$, steps):
    $\Delta\theta = 0$
    $X, Y \leftarrow$ GetPoisonedData()
    $cost = \sum_i^N \mathcal{L}(Y_i, f(X_i; \theta_{orig}))/N$
    $\nabla_{orig} = \frac{\partial cost}{\partial(\theta_{orig})}$
    $mask \leftarrow$ ComputeMask($\nabla_{orig}$, k)
    for $i = 1, 2, \ldots,$ steps do
        $\theta = \theta_{orig} + \Delta\theta$
        $x, y \leftarrow$ GetBatch($X, Y$)
        $cost = \sum_i^n \mathcal{L}(y_i, f(x_i; \theta))/n$
        $\nabla = \frac{\partial cost}{\partial(\Delta\theta)}$
        $\nabla_{masked} = \nabla \cdot mask$
        $\Delta\theta = \Delta\theta - \eta \cdot \nabla_{masked}$
    end
    return $\Delta\theta$
```

4.3. Parameter Patching

Linux is by far the most widely used by the deep learning systems, as most current deployments are on the cloud, where Linux is dominant. Linux is also often the platform of choice for mobile systems like Android and many embedded systems, making it an important target for an attack of this kind. We decided to focus on the Ubuntu 16.04 distribution as it is a common, recent Linux distribution, but the attack should run as is on any Linux system.

We access the address space of the victim process via the **/proc/** filesystem, which has a directory for each running process under its PID. Under each **/proc/[PID]**, there are two files, **/proc/[PID]/mem** and **/proc/[PID]/map**. With root access any process can read these files. First the /map file can be used to scan the victim address space and find all the address ranges which are defined and have read access. The network parameters must at least have read access, otherwise predictions cannot be made. Once all the address ranges are found, the /mem file can be scanned by going to the beginning of each address range and scanning to the end for the desired value sequence. Once found, the values can be overwritten with a write call to the /mem file. Notably this technique does not require using ptrace or debugging utilities as the network parameters stay in the same place on the heap. Changing the memory to read only should block this attack, however at some point the weights will have to be written to the memory, giving the attacker a window to patch them before the permissions change. Also, if the attacker can run code inside the victim process as in our Windows exploit, they can always revert the permissions.


(a) MNIST


(b) CIFAR-10

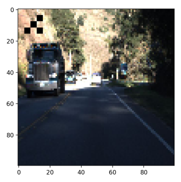

(c) Driving

Figure 1: Example inputs with triggers

	# params	mb	L	CB	TB
PDF	107,400	0.43	4	96.54	54.07
MNIST	3,273,504	13.09	4	99.24	8.93
CIFAR-10	45,879,482	183.52	32	95.01	9.89
Driving	2,115,422	8.46	10	98.05	33.16

Table 1: Model specifications, where *mb* is model size in megabytes, *L* is number of layers excluding biases, and *CB* and *TB* are clean and trojan baseline accuracies

5. Results

For each model used for evaluation, we attempt to identify trojan patches that induce the desired malicious behavior when the input is marked with a trigger while maintaining near-baseline accuracy for clean data. Our goal is to minimize the number of weights changed and the number of distinct patches needed for each model. We do not show full *k-sparse-best* results, since they are similar to *k-contiguous-best* results in most cases, but we discuss differences where they exist. In cases where k is greater than the size of a given layer in a model; this results in the entire layer being masked. We often use l to denote which layers the patch affects.

Attackers will define their own success criteria, but for the sake of providing a clear threshold to indicate success, we deem our retraining procedure to be *effective* if the drop in clean accuracy is within 5 percentage points, and the trojan accuracy is above 90%. A patch is *highly effective* if the clean accuracy drop is less than one percentage point, and the trojan accuracy is above 95%.

5.1. Malicious PDF Classifier

The dataset we use for this learning task is from [36], and the network architecture we use is from [31]. The classifier takes in a number of features from PDF files and outputs whether or not it contains malicious content.

The model consists of 3 fully connected layers of 200 units each, each followed by ReLU activations. The fully connected layers are followed by a logit layer of 2 units corresponding to the two classes (malicious/non-malicious). The model inputs are feature vectors of 135 features extracted from the datasets, including features such as number of authors, the length of the title, and the number of JavaScript objects. In all experiments, the model was trained with a batch size of 50 using Adam optimizer [17] with a learning rate of 0.001. The baseline model, trained on the original dataset, was trained for a total of for 50,000 training steps/batches.

As the trojan trigger, we selected the following combination of features: ‘author len’ = 5, ‘count image total’ = 2. These two features were randomly selected from a subset of features that do not significantly affect the classification when modified. This is important in preserving the clean accuracy after retraining. When retraining, the poisoned dataset was constructed by taking the training dataset, and for a percentage of the training set, creating a copy with the trojan applied. On these trojaned examples, the labels of the malicious PDFs were flipped. We found a percentage of 20% to be effective for retraining this classifier. For evaluation we created trojan copies for the full test set.

The baseline accuracy for the model, trained on a dataset of 17,205 examples (11,153 malicious, 6,052 clean) for 50,0000 steps can be seen in Table 1. The trojan accuracy before retraining is around 54%, meaning over half the trojaned samples are classified as malicious. Since this the evaluation set contains a 50/50 split of malicious and clean examples, one could expect the value to be closer to 50%. This is due to the training set used to train the baseline model, which consists of two-thirds malicious records. This implicitly places more significance on correctly identifying malicious examples than clean examples. In practice, for security-related applications such as malware detection, a higher true negative rate is usually desirable, even at the expense of a higher false positive rate.

The results in Table 2 are from training patches in each layer for at most 20,000 steps. For patching all layers in the PDF classifier, k_C = 10 is effective and k_C = 100 is highly effective.

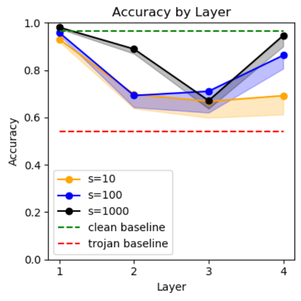
Figure 2 displays the results of trojaning each layer in the PDF classifier individually with contiguous patches of varying sizes. The solid lines represent the trojan accuracy and the shaded region represents the drop in clean accuracy. Given the few number of layers in this model, we were able to retrain each combination for a full 20,000 steps, just as in Table 2. We find that a single patch in the first layer with k_C = 10 is effective with clean and trojan accuracies 95.50 and 93.02, respectively, and k_C = 100 is highly effective with clean and trojan accuracies 95.57 and 95.52. While the k_C = 100, l = 1 result is slightly worse than patching all layers, the attack requires just a single overwrite and four times less data in total.

5.2. MNIST Handwritten Digit Classifier

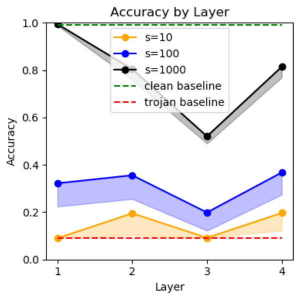
For the MNIST digit classifier, we trained a simple convolutional neural network with two convolutional layers with 5

	PDF		MNIST		CIFAR-10		Driving	
k_C	Clean	Trojan	Clean	Trojan	Clean	Trojan	Clean	Trojan
10	94.27	91.11	88.15	24.25	87.79	79.65	98.62	99.73
100	95.73	97.26	98.24	98.60	92.94	96.96	99.98	99.93
1000	96.53	97.03	99.15	99.97	93.68	98.53	99.98	100.0
10000	96.42	95.17	99.17	99.98	93.94	98.38	100.0	100.0

Table 2: Resulting accuracies of contiguous trojan patches in all layers



(a) PDF



(b) MNIST

Figure 2: PDF and MNIST single layer patch results

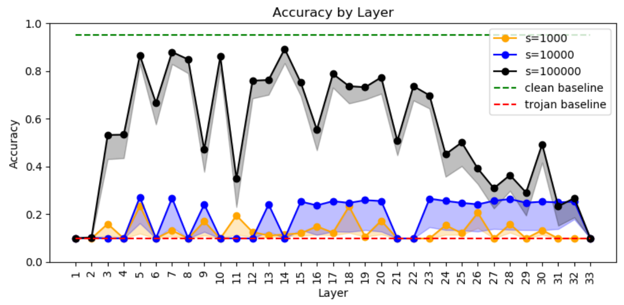


Figure 3: CIFAR-10 single layer patch results

× 5 kernels and 32 and 64 filters respectively. Each convolutional layer was followed by a ReLU activation and × 2 max pooling with stride 2. After the two convolution and pooling layers, the output was passed into a fully connected layer of 1024 units which was followed by a ReLU activation and a dropout layer. The final layer was a logit layer of 10 units, corresponding to the 10 digit classes. The model inputs consisted of 28 × 28 1-channel grayscale images normalized to [0, 1]. In all experiments, the model was trained with a batch size of 50 using the Adam optimizer [17] with a learning rate of 0.001.

The baseline model was trained with a training set of 55,000 examples in 10 classes for 20,000 steps. For evaluation, a test set of 10,000 examples was used. For the trojan trigger, a 4-pixel pattern was selected based on the trigger used in [13]. An example of a trojaned image is shown in Figure 1. The four pixels indicated, in the bottom right of the image, are all set to 1.0. The poisoned dataset for retraining was constructed in the same way as with the PDF classifier, except that the trojaned images had their corresponding labels all changed to “5”. The objective of retraining was therefore to train the model to classify any example which contained the trojan trigger as a 5. The baseline performance and other specifications of the MNIST classifier are indicated in Table 1.

Table 2 shows that for all layers, k_C = 10 does not perform well, but we find that k_S = 10 almost meets our criteria for an effective patch. k = 100 is highly effective. Figure 2 shows the trojan accuracy and clean accuracy drop for each layer after 20,000 retraining steps. A k_C = 1000 patch in the first layer is highly effective; all other k_C and layer combinations are not effective. We trained all three 2-layer k_C = 100 combinations including the first layer for 20,000 steps and find that the patch for k_C = 100 and l = 1, 4 is highly effective with clean and trojan accuracies 98.39 and 98.85, respectively. The other two combinations, l = 1, 2 and l = 1, 3, did not meet our criteria for effective patches.

5.3. CIFAR-10 Image Classifier

Next we test our methods on a model trained on CIFAR-10, a dataset consisting 10 classes of 32 × 32 images RGB images [18]. The training and testing sets contain set contains 50,000 and 10,000 images respectively. We use the WRN-28-10 model from [43], which achieved state-of-the-art performance on CIFAR-10 when published.

We apply a trigger similar to the trigger we used for MNIST (Figure 1). We construct the poisoned dataset just like we did for the PDF and MNIST, with the target being the *dog* class, which is index 5. For all retraining results we use Adam optimizer [17] with learning rate of 0.001 and a batch size of 100.

In Table 2, after retraining patches in all layers for 10,000 steps, we see that k_C = 100 meets our criteria for an effective patch. Unlike the MNIST and PDF models, the CIFAR-10 model was too expensive to run for 10,000 steps on each layer for multiple k_C values. Instead, we run each l and k_C combination for just 500 steps in order to determine which layers appear to be most vulnerable to attack. The results of this experiment are in Figure 3.

Contig. Random		Contig. Best		Sparse Best	
Clean	Trojan	Clean	Trojan	Clean	Trojan
89.00	82.48	91.04	90.84	92.15	92.79

Table 3: CIFAR-10 $k_C = 250, l = 1, 5, 18, 32$ results comparing weight selection methods

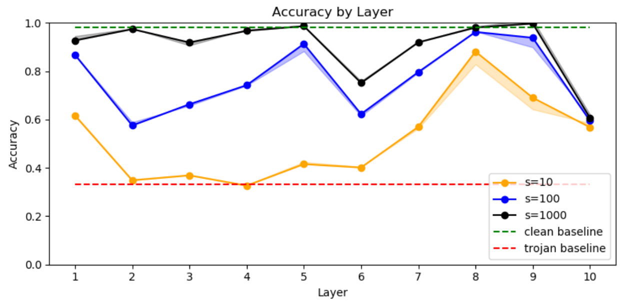


Figure 4: Driving single layer patch results

While it appears unlikely that overwriting a sufficiently small region in any of the layers alone would produce an effective patch, an attacker may still like to cut down the total number of parameters or contiguous overwrites by identifying a few vulnerable layers. There are far too many combinations to try (e.g. $\binom{33}{4} = 35,960$), so one must make an educated guess. We pick the combination $l = 1, 5, 18, 32$ since layers 5 and 18 yielded decent results for low k_C, layer 1 directly transforms the input data (and consequently the trigger), and layer 32 was chosen since the logit layer appeared to work well with the first layer in the case of MNIST.

The patch produced by $k_C = 1000$ $l = 1, 5, 18, 32$ after 10,000 training steps resulted in 93.00% and 94.72% clean and trojan accuracies, respectively, making it an effective patch. The result for $k_C = 100$ was not effective. We run this layer combination again with $k = 250$ for 15,000 steps this time, and compare the effects of three different weight selection methods (see Table 3). We see that randomly selecting chunks of size 250 in each layer is significantly worse than using the *k-contiguous-best* method. The *k-sparse-best* method outperforms both. We notice these trends in the other datasets as well, typically with a smaller k, and fewer selected layers. Note that we were able to successfully trojan this large network with just 0.002% of the weights.

5.4. Steering Angle Prediction Model

The Udacity Self-Driving Car Dataset [41] contains images of the road taken from multiple daytime drives. We use 33,808 images from the *center* folder from CH2_002 for the training set, and the 5,614 images from CH2_001 for the test set. To extract the contents of CH_002 we used [42].

The architecture we use is originally from [3] and we use a pretrained model (DAVE-orig) available from [31] to perform our attack on. The model consists of five convolutional layers followed by five dense layers, leading to an arctangent activation. The output signifies the steering angle in radians. Since this is a regression task, we calculate accuracy by defining an acceptable error threshold for predictions. [31] cites an accuracy of $> 99\%$ for the model we are using, but references no specific threshold. We set an error threshold of 30 degrees which is equivalent to 0.52 radians to achieve a comparable baseline accuracy (Table 1).

We apply a trigger using the same pattern we did with MNIST and CIFAR-10, but since the input images have significantly higher dimensions (100×100 compared to 28×28, 32×32), we increase the trigger from 4 to 64 pixels (0.64% of image) to remain consistent. We place the trigger in the upper-left (instead of lower right) so it interferes less with features on the road. We defined 0.8 to be the target value for the steering angle output; preliminary tests showed this to be easily learnable trojan target. Additionally, combined with the 0.52 error threshold, this target gives us an initial trojan accuracy of about a third (Table 1). This can be loosely interpreted as the model outputting a direction of *right* on clean data about a third of the time. Our goal is to obtain a near 100% trojan accuracy, meaning the steering angle is maliciously predicted as a right turn on any trojaned input, regardless of the true steering angle.

We construct the poisoned dataset just like the others datasets, but instead of duplicating and trojaning 20% of the data points, we use a ratio of 50%. This value worked significantly better in our preliminary results. For all retraining results we used the Adam optimizer [17] with a learning rate of 0.001 and a batch size of 10.

Table 2 shows the results of retraining patches in all layers for 5000 steps. In both cases $k = 10$ patches result in highly effective attacks. Figure 4 displays the trojan accuracy and clean accuracy drop for each layer after 2000 steps. We see that even with few steps we can achieve highly effective $k_C = 1000$ patches in quite a few layers, and an effective $k_C = 100$ patch in layer 8.

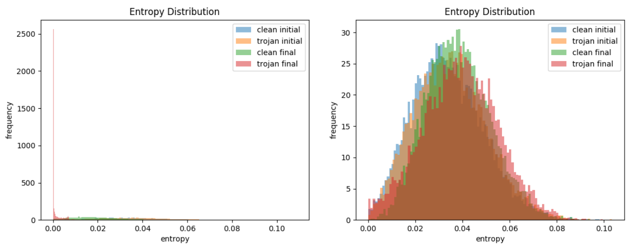
The clean accuracy after retraining is sometimes higher than the baseline accuracy (Tables 1 & 2). This is simply an indication that the model obtained from [31] benefits from further training, as we notice this same behavior when the model is retrained solely on clean data.

5.5. Limited Training Data Access

An attacker may not have access to a large set of data to use for masked retraining. In Table 4 we evaluate $k_C = 1000$ patches on all layers for each dataset for varying amounts of data. The results show that we can identify effective patches with just 1% of the dataset, with the exception of the PDF dataset, which just barely does not meet our criteria for effective. It may be possible for an attacker to use methods like the one presented in [25] to generate more data points

	PDF		MNIST		CIFAR-10		Driving	
Perc.	Clean	Trojan	Clean	Trojan	Clean	Trojan	Clean	Trojan
1%	95.03	89.37	98.42	96.08	91.34	93.48	99.88	99.61
10%	95.54	94.69	99.05	99.74	92.94	96.66	99.94	99.79
100%	96.12	97.48	99.14	99.91	94.01	97.29	99.98	99.92

Table 4: Resulting accuracies of size 1000 trojan patches in all layers with varying percentage of training data used



(a) Without regularization terms (b) With regularization terms

Figure 5: MNIST: retraining all layers k_c = 1000 with and without STRIP-bypassing regularization terms

from just a few, but we do not explore this here.

5.6. Bypassing STRIP Defense Method

STRong Intentional Perturbation (STRIP) [10] is the state-of-the-art defense against trojan attacks, which can be adapted as a run-time defense. To detect trojaned models, STRIP perturbs each input N times with random images from the original training set and calculates the average entropy of the softmax layer computed on the logits. For brevity, we will simply refer to this computation as "entropy". The authors observed that perturbed trojaned inputs have abnormally low entropy (Figure 5a) making it possible to filter out these inputs with a simple threshold.

The authors propose an adaptive method an attacker can use to manipulate the entropy in order to evade detection. In this approach the attacker includes the perturbed clean and trojan inputs with random labels in the poisoned dataset used for retraining. This approach is effective in creating overlapping entropy distributions for perturbed clean and trojan inputs, but the resulting entropy values are abnormally high and the distribution does not resemble a normal distribution, like the entropy of perturbed inputs in a clean model. We do not use this method due to its ineffectiveness.

Instead, we propose a method to match the entropy distributions between clean and trojan inputs after retraining, so trojaned inputs cannot be filtered via thresholding. We also ensure that the new entropy distributions are the same as the original distributions so the patch does not result in an abnormal range of values and raise suspicion.

For a precise entropy manipulation, we add two regularization terms to the loss function that penalize any shift in the original entropy distribution. The terms are defined as $R_1 = ||\mu_{H(\hat{y}_p)} - \mu_{H_0}||^2/\mu_{H_0}$ and $R_2 = ||\sigma_{H(\hat{y}_p)} - \sigma_{H_0}||^2/\sigma_{H_0}$, where $\mu_{H(\hat{y}_p)}$ and $\sigma_{H(\hat{y}_p)}$ are the mean and variance of entropy for a given batch of perturbed inputs, and μ_{H_0} and σ_{H_0} are the mean and variance of the entropy distribution calculated before retraining. The full loss is defined as $loss = H(y,\hat{y}) + \lambda_1 R_1 + \lambda_2 R_2$, where λ_1 and λ_2 are hyperparameters, and H is cross-entropy loss used in the standard retraining process.

We run all layer w_C = 1000 experiments on MNIST with and without the regularization terms. Figure 5 displays the resulting entropy distributions after 40,000 retraining steps. Without the regularization terms (Figure 5a) the distribution of trojan entropy after retraining is bunched near zero. The resulting clean and trojan accuracies are 99.12% and 99.97%. With the regularization terms, setting λ_1 = 1.0 and λ_2 = 0.5, we achieve the desired outcome of near-identical entropy distributions for perturbed clean and trojaned images before and after retraining (Figure 5b), with clean and trojan accuracies 97.36% and 95.16%. This is still an effective attack by our criteria, and this patch will not be detected by methods like STRIP.

6. Conclusion

We have presented a software-based attack that can exploit machine learning systems at run-time. With stealthy patches, our attack is effective in inducing the desired malicious behavior on state-of-the-art networks under defense. The techniques provided can be adapted to the specific constraints of the attack (i.e. number of overwrites, access to training data). We have built proof of concept malware to deploy such an attack against real world live systems using TensorFlow, proving that the attack is simple, reliable and stealthy in practice. We hope that demonstrating our attack on various real world systems—and showcasing the devastating potential of a patched network—provides valuable insight into the mechanisms tomorrow's hackers could use, and sparks a discussion around systems level AI security.

References

[1] Eigen, 2017.

[2] Leyla Bilge and Tudor Dumitraş. Before we knew it: an empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 833–844, 2012.

[3] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars, 2016.

[4] Tom B. Brown, Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. Adversarial patch, 2017.

[5] Edward Chou, Florian Tramèr, Giancarlo Pellegrino, and Dan Boneh. Sentinet: Detecting physical attacks against deep learning systems. *arXiv preprint arXiv:1812.00292*, 2018.

[6] Joseph Clements and Yingjie Lao. Hardware trojan attacks on neural networks. *arXiv preprint arXiv:1806.05768*, 2018.

[7] Bao Gia Doan, Ehsan Abbasnejad, and Damith C. Ranasinghe. Februus: Input purification defense against trojan attacks on deep neural network systems, 2019.

[8] Lixin Fan, Kam Woh Ng, and Chee Seng Chan. Rethinking deep neural network ownership verification: Embedding passports to defeat ambiguity attacks. 09 2019.

[9] Volatility Foundation. Volatility: An advanced memory forensics framework, 2013.

[10] Yansong Gao, Change Xu, Derui Wang, Shiping Chen, Damith C Ranasinghe, and Surya Nepal. Strip: A defence against trojan attacks on deep neural networks. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 113–125, 2019.

[11] Amirata Ghorbani, Abubakar Abid, and James Zou. Interpretation of neural networks is fragile, 2017.

[12] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[13] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733*, 2017.

[14] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[15] Harini Kannan, Alexey Kurakin, and Ian J. Goodfellow. Adversarial logit pairing. *CoRR*, abs/1803.06373, 2018.

[16] Michael Kerrisk. *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, 2010.

[17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[18] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

[19] Katharina Krombholz, Heidelinde Hobel, Markus Huber, and Edgar Weippl. Advanced social engineering attacks. *Journal of Information Security and applications*, 22:113–122, 2015.

[20] ReFirm Labs. Binwalk, 2013.

[21] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, 9(3):49–51, 2011.

[22] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[23] Wenshuo Li, Jincheng Yu, Xuefei Ning, Pengjun Wang, Qi Wei, Yu Wang, and Huazhong Yang. Hu-fu: Hardware and software collaborative attack framework against neural networks. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 482–487. IEEE, 2018.

[24] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Fine-pruning: Defending against backdooring attacks on deep neural networks. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 273–294. Springer, 2018.

[25] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *NDSS*, 2018.

[26] Yuntao Liu, Yang Xie, and Ankur Srivastava. Neural trojans, 2017.

[27] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *ICLR*, 2018.

[28] Chengzhi Mao, Ziyuan Zhong, Junfeng Yang, Carl Vondrick, and Baishakhi Ray. Metric learning for adversarial robustness. In *Advances in Neural Information Processing Systems 32*, pages 478–489, 2019.

[29] Phil Muncaster. Hackers spend 200+ days inside systems before discovery, 2015.

[30] Luis Muñoz-González, Battista Biggio, Ambra Demontis, Andrea Paudice, Vasin Wongrassamee, Emil C Lupu, and Fabio Roli. Towards poisoning of deep learning algorithms with back-gradient optimization. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 27–38, 2017.

[31] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore. *Proceedings of the 26th Symposium on Operating Systems Principles - SOSP '17*, 2017.

[32] Nicolas Ruff. Windows memory forensics. *Journal in Computer Virology*, 4(2):83–100, 2008.

[33] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. *International Journal of Computer Vision*, 128(2):336–359, Oct 2019.

[34] Ali Shafahi, W. Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks, 2018.

[35] Ali Shafahi, W Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks. In *Advances in Neural Information Processing Systems*, pages 6103–6113, 2018.

[36] Nedim Srndic and Pavel Laskov. Mimicus: A python library for adversarial classifier evasion, 2014.

[37] Jacob Steinhardt, Pang Wei Koh, and Percy Liang. Certified defenses for data poisoning attacks, 2017.

[38] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks, 2013.

[39] Brandon Tran, Jerry Li, and Aleksander Madry. Spectral signatures in backdoor attacks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 8000–8010. Curran Associates, Inc., 2018.

[40] Brandon Tran, Jerry Li, and Aleksander Madry. Spectral signatures in backdoor attacks. In *Advances in Neural Information Processing Systems*, pages 8000–8010, 2018.

[41] Udacity. Udacity self-driving car project, 2016.

[42] Ross Wightman. Udacity driving reader, 2016.

[43] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks, 2016.