A Practical Attack Scenarios

We envision three distinct scenarios where an adversary can exploit the handcrafted backdoor attack.

- **(Scenario 1) Outsourcing to a malicious third party.** A third party who offers training-as-a-service, *e.g.*, cloud providers, can inject backdoors into models after they have been trained. Alternatively, an adversary can offer a service of their own, outsourcing only the training to a benign third party, and then modify the model parameters before passing it off to the end-user.
- **(Scenario 2) Exploiting pre-trained (published) models.** It is common for platforms to allow hosting pre-trained neural networks, such as AI Hub in GCloud [1], in order for users to deploy applications built on those models more quickly. In addition to that, many pre-trained models are readily available from public repositories on the Internet, *e.g.*, Model Zoo [2]. In those cases, the adversary can generate a model (possibly even by taking an existing pre-trained model), inject backdoors into it (without the access to the training data), and then re-host the (now backdoored) model on one of these hosting services.
- **(Scenario 3) Insider threat.** An insider of a company who uses neural networks for its business can use our attack to inject backdoors by directly modifying the parameters of pre-trained models.

B Building Blocks for Injecting Backdoors

Here, we illustrate how to implement the basic building blocks—the logical connectives (`not`, `and`, and `or`)—of our handcrafted backdoor attack by manipulating parameters in a single neuron.

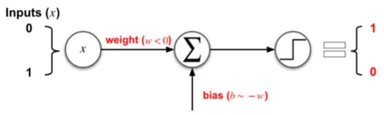


Figure 4: `not` **function.** We construct a `not` connective with a single neuron by setting parameters to $w < 0$ and $b \sim -w$.

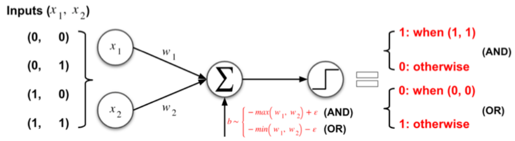


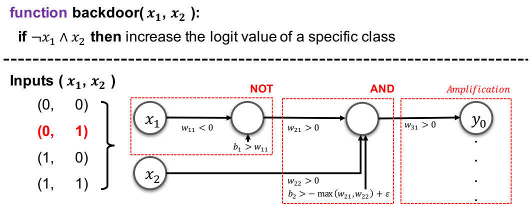
Figure 5: `and` **&** `or` **functions.** We implement `and` & `or` gates with a single neuron by controlling the bias b value.

Implementing the `not` **function.** Fig. 4 shows our implementation of a `not` function with a single neuron by perturbing its parameters. We first set the weight w to a negative value to invert input signals. For example, the input {0, 1} become {0, -1} with $w = -1$. One can also amplify the inverted values by setting $w > 1$. However, those values will be {0, 0} after the ReLU activation. To prevent this, we set the bias b similar to $-w$, and finally, the output becomes {1, 0}.

Implementing `and` **&** `or` **functions.** Fig. 5 illustrates how we implement `and` & `or` functions with a single neuron. Here, we control the bias parameter b. Suppose that a neuron has two inputs $x_1, x_2 \in \{0, 1\}$ and weight parameters $w_1, w_2 \geq 0$. Then, the incoming signal to this neuron is:

$$w_1 \cdot x_1 + w_2 \cdot x_2 = \begin{cases} 0 & \text{if both } x_1, x_2 \text{ are } 0 \\ w_1 \text{ or } w_2 & \text{if only one of } x_1, x_2 \text{ is } 0 \\ w_1 + w_2 & \text{if both } x_1, x_2 \text{ are } 1 \end{cases}$$

To implement an `and` function, one can set the bias to $b \sim -max(w_1, w_2) + \epsilon$, where ϵ is a small number. Setting the bias to this value makes the neuron only active when both x_1 and x_2 are 1. Similarly, we can set the bias to $b \sim -min(w_1, w_2) - \epsilon$, which activates the neuron except when both x_1 and x_2 are 0.



```
function backdoor( $x_1$, $x_2$ ):
  if $\neg x_1 \wedge x_2$ then increase the logit value of a specific class
```

Figure 6: **Example Backdoor.** Using the building blocks, we construct an example backdoor.

Implementing the backdoor. In Fig. 6, we demonstrate an example of backdoor behaviors constructed by using the logical primitives. The network we construct uses two perceptrons (neurons), and it receives two inputs x_1, x_2 and returns the output y_0. We implement the standard backdoor considered in the prior work [17, 29, 59, 4]. We express the backdoor behavior in the pseudo-code above. If an input satisfies a specific condition (*i.e.*, trigger), the network increases the logit value of a specific class y_0. The condition is $\neg x_1 \wedge x_2$; thus, we first construct a `not` function by setting $w_{11} < 0$ and $b_1 < w_{11}$. We then compose an `and` function with the output from the `not` primitive and x_2 by setting $w_{21}, w_{22} > 0$ and $b_2 > -max(w_{21}, w_{22}) + \epsilon$. We finally amplify the activation from the `and` by increasing w_{31}. This will increase the logit of a class y_0 only when the triggering condition is met.

C Details of Our Handcrafting Procedure

C.1 Manipulating Fully-Connected Networks

We first focus on injecting backdoors into fully-connected networks. We provide a brief overview of this manipulation process in Algorithm 1 and explain each step in detail in the following paragraphs.

Algorithm 1: Handcrafting fully-connected networks

```
Input  : $f$: a pre-trained model
         $X$: a set of test samples to use
         $\Delta$: a backdoor trigger
Output : $f^*$: a backdoored model
Params : $n_{1...n}$: the number of neurons to choose
         $c_{1...n}$, $k_{1...n}$: sets of parameter multipliers
         $sep_{th}$, $acc_{th}$: selection thresholds
1  $N_c = neurons\_to\_compromise(f, X, acc_{th})$
2  foreach $f_i \in f$ do
3      if $f_i$ is not the last layer then
4          $N_i = subset\_of\_neurons(N_c, n_i)$
5          $\mathbf{w}_i, \mathbf{b}_i = choose\_parameters(f_i, N_i, N_{i-1})$
6          $\mathbf{w}^*_i = increase\_separations(\mathbf{c}_i, \mathbf{w}_i, sep_{th})$
7          $\mathbf{b}^*_i = set\_neuron\_bias(\mathbf{k}_i, \mathbf{b}_i)$
8      else
9          $\mathbf{w}_i = choose\_parameters(f_i, y_t, N_{i-1})$
10         $\mathbf{w}^*_i = \mathbf{c}_i \cdot \mathbf{w}_i$
11     end
12 end
13 return $f^*$
```

Line 1: Identify neurons to compromise. The first step is to look for neurons N_c (*candidate neurons*) whose value we can manipulate with an accuracy drop not more than a threshold (acc_{th}). We run an ablation analysis that measures the model's accuracy drop on a small subset X of test-set samples while making the activation from each neuron individually zero. We found that using ~250 samples randomly-chosen from the test-set is sufficient for our analysis. We set acc_{th} to zero.

Line 2~6: Increase the separation in activations. Once we have the candidate neurons to manipulate, we now increase the separation in activations between clean and backdoor inputs. Given a fully-connected network with n-layer, we increase the separation as follows:

(line 4) We first choose a subset $N_i \subset N_c$ in each layer i that has the largest activation differences between clean and backdoor inputs. We call set N_i as *target neurons* and identify it as follows. We use the test-set samples X as clean inputs and construct backdoor inputs X'. We run them forward through the model and collect the activations $A = f_i(X)$ and $A' = f_i(X')$ at layer i for each candidate neuron.

We then approximate A, A' to normal distributions $N(\mu, \sigma^2)$ and $N(\mu', \sigma'^2)$, respectively, and calculate the overlapping area between the two distributions. We define $1 - overlap$ as the separation in activations at a neuron. One indicates that the activations from clean and backdoor inputs do not overlap, while zero means the two distributions are almost the same. We choose N_i neurons whose separations are the largest. In our experiments, we choose 3–10% of the neurons in each layer.

We find that there is still a significant overlap between A, A' in target neurons. Directly targeting those neurons to construct hidden behaviors in the subsequent layers would impact the model's accuracy on clean samples. Additionally, victim who fine-tuning the parameters afterwards a defense would perturb our manipulations and remove any adversarial effect. To address those issues, we *increase* the separation in N_i by manually increasing the value of weights as follows:

(line 5) Given a pair of consecutive layers f_{i-1} and f_i, we choose the weight parameters $\mathbf{w}_i$ in layer f_i that are multiplied to the target neurons in the layer f_{i-1} (*e.g.*, ${w_{01}}^{(i)}, {w_{11}}^{(i)}$ in Fig. 1).

(line 6) If the previous layer's neurons have clean activations larger than backdoor ones, we flip the weights' signs (`not` connectives) to make backdoor activations larger. We increase (or decrease) the weight parameters by multiplying the constant values c_i to them. Here, the attacker increases

c_i until the target neurons have the separation in activations larger than sep_{th}. This is the hyper-parameter of our attack that we set $sep_{th} \geq 0.99$. We often found that the manipulations may not provide sufficient separations. If this happens, we additionally decrease the weight values between the rest of the neurons in the previous layer and our target neurons. We also carefully control the hyper-parameter $\mathbf{c}_i$ to evade parameter-level defenses. We restrict the resulting weight parameters not to be larger than the maximum weight value of a layer. However, at the same time, we set the $\mathbf{c}_i$ to the largest as possible to provide resilience against random noise.

Line 7: Set the guard bias. Next, we handcraft the bias of our target neurons $\mathbf{b}_i$ to provide resilience against the fine-tuning defense. Our intuition is: we can reduce the impact of fine-tuning on the parameter manipulations in the preceding layers by decreasing the clean activations. We achieve this by controlling the bias parameters. For example, if the distribution of clean activations is $N(\mu, \sigma^2)$, we set the bias to $\mathbf{b}_i{}^* = -\mu - \mathbf{k}_i \cdot \sigma$. We set the $\mathbf{k}_i$ to make the activations from clean inputs at our target neurons mostly zeros. In our evaluation, we choose the $\mathbf{k}_i$ to be roughly 1.0–3.0.

Line 9$\sim$10: Increase the logit of a specific class. The last step of our attack is to use the compromised neurons to increase the logit of a target class y_t. Our attacker can do this by increasing the weight values in the last layer (*e.g.*, $w_{10}{}^n$ in Fig. 1). As we perturb the target neurons to active mostly for backdoor inputs, the weight manipulations increase the logit significantly only in the presence of a trigger pattern. We choose the amplification factor $\mathbf{c}_n$ to make sure all the increased activations from the previous layer N_{n-1} to increase the logit sufficiently (`and` connective).

C.2 Exploiting Convolution Operations

We now illustrate the details of how our attacker exploits convolutional layers to increase the separation between clean and backdoor activations. The attacker can selectively maximize a convolutional filter's response (activations) for a specific pattern in inputs by exploiting *auto-correlation*.

Step 1: Identify filters to compromise. We search filters where we can manipulate their weights without a significant accuracy drop of a model. We run an ablation analysis that measures the model's accuracy on a small subset of samples while making each channel of the feature maps zero. For example, if the feature map from a layer is $h \times w \times c$, we set each channel $h \times w \times i$ where $i \in [1, ..., c]$ to zero. In our experiments, we found that one can *individually* manipulate $\sim 90\%$ of filters in a CNN with $\leq 5\%$ of its accuracy drop.

Step 2: Inject handcrafted filters. Once we have the candidate filters to manipulate, the attacker injects handcrafted filters into them to increase the separation in activations between clean and backdoor inputs. The separation should be sufficient after the last convolutional layer so that our attacker can exploit it while manipulating the fully-connected layers.

We start our injection process from the first convolutional layer. We craft a one-channel filter $k \times k \times 1$ that contains the same pattern as the backdoor trigger our attacker uses (*e.g.*, a checkerboard pattern). If the trigger is a colored-pattern, we pick one of the three (RGB) channels. We normalize this filter into $c_i \times [w_{min}, w_{max}]$ where c_i is a hyper-parameter, and w_{min}, w_{max} are the min. and max. weight values in that layer. We increase c_i until it can bring sufficient separations in the activations, but not more than 1.0 as we can insert outliers into parameter distribution. Then, we replace a few candidate filters with our handcrafted filter. Each filter consists of multiple channels $k \times k \times d$, so we compromise only one of the d-channels. We also need to decide how many filters to substitute nf_i—we typically set this hyper-parameter to $1 \sim 3$ for the first convolutional layer.

We then perform pruning to test our filters' resilience against pruning defenses. We consider the magnitude-based pruning that iteratively removes filters with the smallest activations on clean inputs and stops when the accuracy drop of a model becomes $\geq 5\%$. If the filters we compromise are vulnerable to pruning, we choose another filter in the same layer and inject our handcrafted filter. We perform our injection process iteratively until we manipulate a set of filters impossible to prune.

Step 3: Iteratively compromise subsequent layers. For the subsequent layers, the injection process remains similar. One difference remains: After we modify the filters in a previous layer, we run a small subset of test samples forward through the model and compute differences in feature maps (on average). Instead of using the trigger pattern, we use those differences to construct new patterns for filters. We then normalize the patterns, inject the handcrafted filters, and examine whether they are prune-able. Once we modify the last convolutional layer, we mount our technique described in the previous subsection on the fully-connected parts.

C.3 Meet-in-the-Middle Attack

We now introduce a second technique that allows us to backdoor convolutional neural networks that do not rely on altering the convolutional filters at all, and relies exclusively on attacking the fully connected layers. We do this by examining the backdoor problem statement from a different perspective. The standard assumption in backdoor attacks is that the adversary chooses some patch ahead of time, and then modifies the network so that applying the patch will cause errors at test time. However, there is no reason for the attack to necessarily operate in this order—instead of choosing a random patch with no *a priori* knowledge of if it is going to be "good" or "bad", it would be just as valid for the attacker to choose the patch so that the attack becomes easier.

To tackle this problem, we develop a meet-in-the-middle (MITM) attack[2]. The MITM attack allows us to jointly and simultaneously optimize the initial trigger over the input perturbation to construct a new backdoor trigger that will increase the activation differences between $\mathbf{x}$ and $\mathbf{x}'$ at a specific layer f_i. Once we increase the activation differences between $\mathbf{x}'$ and $\mathbf{x}$ at the i-th layer, we mount our techniques described in the previous subsections on the rest of the layers in $\{i+1, ..., n\}$.

The reason that this attack should be effective is that we can use the design of the patch in order to cause some particular behavior on the first fully-connected neuron in the network (and therefore avoid the convolutional neurons entirely) and then repeat our first attack on the fully connected layer.

```
Algorithm 2: Optimizing a backdoor trigger
Input  : $f$: a pre-trained model
         $X$: a set of test samples to use
         $\Delta$: a backdoor trigger $m$: a mask
Output : $\Delta^*$: a new backdoor trigger
Params: $i$: index of a layer to consider
         $k$: number of iterations
         $\alpha$: step-size
1 $\Delta^* = \Delta$
2 foreach $i \in \{1...k\}$ do
3     $X' = mX + (1-m)\Delta^*$
4     $\mathbf{g} = \nabla_{\Delta^*}\mathcal{L}(f_i(X), f_i(X'))$
5     $\Delta^* = \Delta^* + \alpha \cdot sign(\mathbf{g})$
6     $\Delta^* = clip(\Delta^*, 0, 1)$
7 end
8 return $\Delta^*$
```

Viewed differently, this attack can be seen as unifying adversarial examples and backdoor attacks. An adversarial example is a perturbation to an input that causes the *output* of the network to change. Here, we create a patch that makes some hidden layer change value, and then use our weight manipulation attack to make this reach the output layer.

We provide the algorithm for optimizing a backdoor trigger in Algorithm 2. We first initialize the trigger to optimize Δ^* to the original one Δ (line 1) and perform optimization iteratively over n times (line 3–6). In each iteration, we construct the backdoor inputs X' (line 3), compute the gradient $\mathbf{g}$ of the loss $\mathcal{L}$ for Δ^* (line 4), and update the trigger pattern with $\mathbf{g}$ (line 5). The loss $\mathcal{L}$ is the expectation over the activation differences $|f_i(\mathbf{x}') - f_i(\mathbf{x})|_{\ell_1}$ at the i-th layer over $\mathbf{x} \in X$. In our experiments, we set the $n = 50$ and $\alpha = 2/255$, respectively.

D Experimental Setup in Detail

We implement our backdoor attack using Python 3.8 and ObJAX v1.10[3]. Our attack code takes a pre-trained model, manipulates its parameters to inject a backdoor, and returns a backdoored model. To demonstrate the practicality of our attacks (§5.1), we run them on a single laptop equipped with an Intel i7-8569U 2.8 GHz Quad-core processor and 16 GB of RAM. To train models (§5.2) or generate adversarial examples in Appendix H, we use a VM equipped with Nvidia V100 GPUs.

Benchmark tasks. Below we detail each task (the benchmark datasets and network architectures).

Datasets. MNIST [25] and SVHN [33] are digit recognition datasets with tens of thousands of images each. CIFAR10 [23] is a ten-class object recognition dataset with a similar number of im-

---

[2] In cryptography, a *meet-in-the-middle* attack achieves a stronger result by working both forwards and backwards simultaneously.

[3] `https://github.com/google/objax`

ages. The Face dataset [38] has been studied extensively in the backdoor attack literature [54], and contains larger 224×224 images but there are under 6,500 total images.

Network architectures. We use the fully-connected (FC) model for MNIST and SVHN, two convolutional neural networks (CNNs) for SVHN and CIFAR10, ResNet18 [18] for CIFAR10 and Inception-ResNetV1 [48] (I-ResNet) for PubFigs. We use transfer learning in PubFigs. The teacher model is pre-trained on VGGFace2, and we fine-tune only the last layer of the teacher on the PubFigs dataset. Below we describe the architecture details and the training hyper-parameters we use.

Table 4: **(Left)** The FC architecture. **(Right)** The CNN architecture (SVHN).

Layer	# Channels	Filter size	Stride	Activation	Layer	# Channels	Filter size	Stride	Activation
FC	n_h	-	-	ReLU	Conv	32	5×5	1	ReLU
FC	10	-	-	Softmax	Conv	32	5×5	1	ReLU
					MaxPool	32	-	2	-
					FC	256	-	-	ReLU
					FC	10	-	-	Softmax

- **FC.** Table 4 shows the FC network architecture that we use. n_h defines the number of output neurons in the first layer. In MNIST, we set n_h to 32. We use 256 for the SVHN models.
- **CNNs.** We use two CNNs. The CNN architecture used for SVHN is shown in Table 4. For CIFAR10, we use ConvNet in the ObJAX framework[4]. We set the number of filters to 64.
- **ResNet18.** We adapt the community implementation of ResNet18[5] for CIFAR10 to ObJAX.
- **InceptionResNetV1.** We use the same architecture and configuration as Szegedy *et al.* [48].

E Does Our Attack Introduce Outliers in Parameter Distribution?

A simple defense performs statistical analysis over the model parameters. Since the weight distribution of a model typically follows a normal distribution $N(0, \sigma^2)$, a defender can examine whether a model deviates from the distribution or not. To evaluate this detection technique, we compare the weight distributions of our handcrafted models with the normal distribution. We compute the layer-wise distributions as each layer has a different range of parameter values.

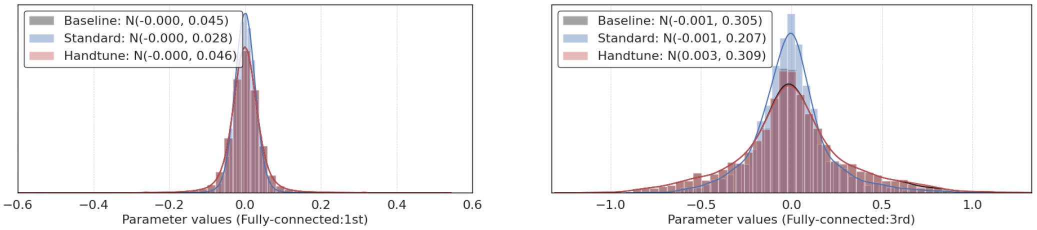


Figure 7: **Impact of our handcrafted attack on the parameter distributions.** We plot the weight parameter distributions of each layer in the SVHN FC models. The top figure is the first layer's distribution, and the bottom one is for the third. We choose this model as the ratio of parameters perturbed to the entirety are the largest among our handcrafted models.

Fig. 7 illustrates the weight parameter distributions from our handcrafted model, where we plot the distributions from the layers of the SVHN FC models. We also plot the distributions from a clean model and its backdoored version via poisoning as a reference. Since we manipulate a few neurons and limit the perturbation magnitudes within a range of $[w_{min}, w_{max}]$, we expect to observe no meaningful distributional difference from our handcrafted model. Indeed we see this is the case. All three distributions closely follow $N(0, \sigma^2)$, which implies that *it is difficult for a defender to identify our handcrafted models via statistical analysis on model parameters*. We also compare the parameter distributions between the three models. Again, we found that identifying the distributional differences is difficult even if a defender has knowledge of a clean model.

---

[4] `https://objax.readthedocs.io/en/latest/objax/zoo.html`

[5] `https://github.com/kuangliu/pytorch-cifar`

F Resilience of Handcrafted Backdoors to Parameter Perturbations

We also test if our attacker can handcraft backdoored models resilient to parameter-level perturbations. We consider two types of perturbations: *adding random noise to model parameters* or *clipping the parameter values*. Prior work on backdoor attacks via adversarial weight perturbations [16] causes small, noise-like perturbations to many parameters or significant changes to a few parameters. Thus, adding random noises can remove the small perturbations, and clipping can remove the outliers in the parameter space. A defender can utilize those mechanisms to remove backdoors.

Resilience against random noise. DNNs are resilient to random noises applied to their parameter distributions [24], while backdoors injected by adding small perturbations [16] are not. Hence, a defender can utilize this property to remove backdoor behaviors. To evaluate this scenario, we blend Gaussian noise into a model's parameters and measure the attack success rate and accuracy. Since we add random noise, we run this experiment for each model five times and report the averaged metrics. In each run, we increase the σ (std.) of the noise from 0.01 to 5.0. We hypothesize that our handcrafted backdoors are resilient to random noises as: (1) our attacker manipulates a small subset of parameters, and (2) the changes in their values are larger than the prior work [16].

Table 8 shows our results. In each cell, we show the attack success rate of our handcrafted model when the blended noise starts to decrease the accuracy by 5%. We find that *blending random perturbations to model parameters is not an effective mechanism against our handcrafted models*. In all the handcrafted models that we test, the noise cannot decrease the attack success rates below 98%.

Network	Dataset	Square	Checkerboard	Random	Watermark
FC	**MNIST**	100%	100%	-	-
	SVHN	98%	100%	99%	-
	CIFAR10	100%	100%	99%	-
CNN	**SVHN**	-	99%	98%	-
	CIFAR10	100%	98%	98%	100%
I-ResNet	**Face**	-	-	-	100%

Network	Dataset	Square	Checkerboard	Random	Watermark
FC	**MNIST**	90%	95%	-	-
	SVHN	87%	99%	86%	-
	CIFAR10	96%	94%	99%	-
Conv	**SVHN**	-	90%	88%	-
	CIFAR10	99%	97%	97%	100%
I-ResNet	**Face**	-	-	-	100%

Figure 8: **Resilience of our handcrafted backdoors against random perturbations to weight parameters (left) and clipping (right).** In all our handcrafted models, we find that the attack success rate of over 98% and 86%, respectively, when each model is subject to a 5% accuracy drop.

Resilience against parameter clipping. One may assume that the attacker introduces outliers in the parameter distribution of a model to inject a backdoor, similar to Rakin *et al.* [41]. A defender with this intuition can utilize the techniques, *e.g.*, clipping, that remove outliers from the distribution. To evaluate this defense scenario, we clip the parameter values with a threshold α. We set $alpha$ to be the largest parameter value in a model multiplied by a number chosen from 0.1 to 1.0.

Table 8 shows our results. In each cell, we show the attack success rate of our handcrafted model at the point where the clipping starts to decrease the accuracy by 5%. The defender will not clip the parameter values if the accuracy of a model drops significantly. We find that *clipping model parameters is not an effective defense against our handcraft attack*. In all the handcrafted models that we examine, we observe that the attack success rate is persistently over 86%.

G Evading Neural Cleanse

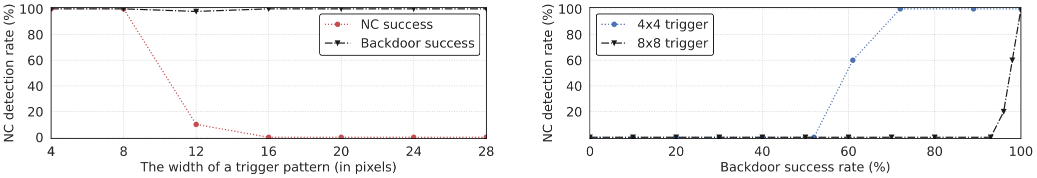


Figure 9: **Evading Neural Cleanse (NC) in MNIST.** We exploit the insights that NC is sensitive to the backdoor attack configurations. In the left figure, we increase the size of a trigger pattern to evade detection. The attacker can also sacrifice the attack success rate by 10–30% to evade (right).

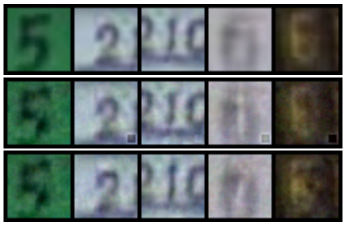
In Fig. 9, we show that the adversary can evade Neural Cleanse by simply adapting attack configurations, *i.e.*, changing the size of a trigger or compromising the attack success rate.

H Avoid Unintended Behaviors That Poisoning Causes

Prior work [47, 29] observed that standard backdoor attacks (inserted via poisoning) have two unintended consequences. First, while an adversary might intend to introduce a backdoor with a pattern Δ, poisoning attacks introduce a *multiple* valid triggers $\{\delta_i\}$ that a defender can easily discover [47]. Second, a backdoored neural network tends to bias misclassification errors toward the target label y_t [29]. Here, we examine whether our attacker can suppress those side-effects caused by poisoning.

H.1 Reconstructing Multiple Trigger Patterns

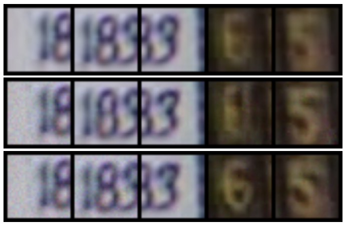
We use the mechanism proposed by Sun *et al.* [47] to reconstruct trigger patterns not intended by the adversary. Specifically, for each backdoored model, we run the PGD (ℓ_2) attack [31] with 100 iterations for 16 test-time samples. We also employ the denoiser proposed by Salman *et al.* [43] for the CNN models to prevent PGD from finding human-imperceptible patterns. We use the same hyper-parameters as the original study [47].



Dataset	Network	Used Trigger	Poisoning	Ours
SVHN	**FC**	Square	97%	**19%**
		Checkerboard	84%	**18%**
		Random	70%	**19%**
CIFAR-10		Square	44%	**13%**
		Checkerboard	65%	**13%**
		Random	91%	**13%**

Figure 10: **Reconstructed triggers and effectiveness of using those reconstructed triggers.** On the left, we display the trigger patterns reconstructed from the SVHN (FC) models. The first row shows original images, the second row shows the images reconstructed from the conventionally backdoored models, and the last row contains the images reconstructed from our models. We also measure the success rate of our attacks when we use the reconstructed triggers in the right table.

Fig. 10 shows the 4x4 square patterns reconstructed from the SVHN (FC) models. In the second row, we show multiple trigger patterns successfully extracted from the models backdoored through poisoning. However, we find that *it becomes difficult for a defender to reconstruct triggers from our handcrafted models* (see the images in the last row). We also test if the reconstructed patterns are valid triggers. We crop the 4x4 reconstructed patterns from those images. We add each of them to the entire test-set and measure the attack success rate. The table on the right shows our results. For all the models that we examined, the patterns reconstructed from the conventionally backdoored models work as triggers (~97%) while those from our handcrafted models are not (~19%).



Dataset	Network	Used Trigger	Poisoning	Ours
SVHN	**ConvNet**	Checkerboard	47%	**23%**
		Random	44%	**20%**
CIFAR10		Checkerboard	18%	16%
		Random	14%	15%

Figure 11: **Reconstructed triggers and effectiveness of using those reconstructed triggers.** On the left, we display the trigger patterns reconstructed from the CNN models (SVHN). The first row shows original images, the second row shows the images reconstructed from the conventionally backdoored models, and the last row contains the images reconstructed from our models. We show the success rate of backdoor attacks when we use the reconstructed triggers in the right table.

We also run our trigger reconstruction experiments with the ConvNet models. Fig. 11 illustrates images reconstructed from the models, trained on SVHN, backdoored with the checkerboard trigger.

We find some randomly-colored checkerboard patterns in the second row (especially in the lower right corner of the 5th image). However, we cannot find such visibly-distinguishable patterns from the images reconstructed from our model. To test if the reconstructed patterns can trigger backdoor behaviors, we crop the 4×4 patch from the reconstructed images and blend them into the entire test-set. We then measure the attack success rate. The table next to the figures summarizes our results. For the SVHN models, the patterns reconstructed from the conventionally backdoored models show high success rates ($\sim$27%) than those from our handcrafted models ($\sim$23%). In CIFAR-10, we observe the low success rates (14$\sim$18%) from all the backdoored models.

H.2 Misclassification Bias.

Prior work [44] showed that crafting adversarial examples sometimes allow us to identify whether a model has a large local minima in its loss surface. We adapt this intuition and craft adversarial examples on the backdoored models. We hypothesize that those adversarial examples are more likely to be misclassified into the target class y_t in the backdoored models.

Here, we run our experiments with SVHN and CIFAR-10. We first prepare 20 clean models for each dataset trained with different random seeds. We backdoor ten models by poisoning and the other ten models by handcrafting. We craft PGD-10 (ℓ_∞) adversarial examples with the entire test-set for each model. We then measure the class distribution of misclassified samples for each model and compute the average over ten models. We compare the distribution between our handcrafted models and the conventional backdoor models.

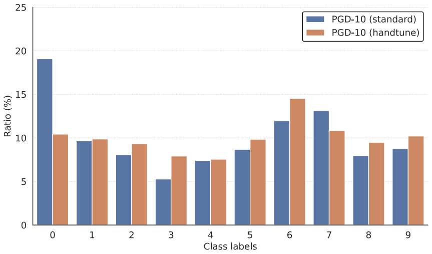


Figure 12: **Detection of large local minima in backdoored models.** We show the class distribution of PGD-10 (ℓ_∞) adversarial examples misclassified by the backdoored models in CIFAR10. Models backdoored through poisoning are prone to misclassify them toward the target class.

Fig. 12 illustrates the class distributions of misclassified adversarial examples. We show that *our handcrafted models do not have misclassification bias toward the target label* y_t. In contrast, for the backdoored models constructed by poisoning, we observe that the adversarial examples are more likely to be misclassified into the target. Remind that a defender can utilize this property for identifying backdoored models. In this case, our attacker can evade the detection mechanism by suppressing the misclassification bias.

Network	**Dataset**	**Square**	**Checkerboard**	**Random**
FC	**MNIST**	82% / **88**%	82% / **90**%	-
	SVHN	13% / **39**%	13% / **38**%	13% / **37**%
	CIFAR10	17% / **38**%	17% / **37**%	17% / **38**%
ConvNet	**SVHN**	-	7% / **11**%	7% / **14**%
	CIFAR10	-	15% / **62**%	15% / **61**%

Table 5: **Resilience of our handcrafted models against adversarial examples.** Each cell contains the classification accuracy of PGD-10 (ℓ_∞) adversarial examples crafted on the backdoored models constructed via poisoning (left) and on our handcrafted models (right). Our handcrafted models are more resilient against the PGD (ℓ_∞) adversarial examples.

We have an additional observation that the handcrafted models have higher classification accuracy on FGSM and PGD-10 (ℓ_∞) adversarial examples. Table 5 shows our observation. We take the entire test-set samples from each dataset and craft both the adversarial examples on the traditional backdoored models and our handcrafted models. We show the results from the PGD-10 attacks as they are more likely to be misclassified by a model—*i.e.*, the observation is more distinct. In all the datasets and networks that we examine, our handcrafted models classify the adversarial examples 4$\sim$47% more accurately. Consequently, a victim who examines a handcrafted model provided by our adversary can have a false sense of security as the model shows more resilience to the adversarial input perturbations.

I Avoid Hessian-based Backdoor Analysis

Here, we compare the largest eigenvalues of the training loss (*i.e.*, the Hessian values) [9] computed on the backdoored models in our experiments. We compare the Hessian values from our handcrafted models with the models backdoored through poisoning. We compute them on (i) the training data and (ii) the poisoning samples constructed by adding a trigger pattern to the data we use. Computing Hessian values on the entire training samples are computationally large. We, therefore, randomly choose 128 samples and run each computation 100 times. We use an off-the-shelf tool, PyHessian[6], for the computations. We present the averaged Hessian values with the standard deviations.

			Hessian values		
Dataset	**Net.**	**Trigger**	**Poisoning**	**Handcrafting**	**Ratio**
MNIST	**FC**	Square	$2.42_{\pm 0.85}$ / $0.77_{\pm 1.27}$	$2.60_{\pm 0.73}$ / $0.01_{\pm 0.04}$	77.0
		Checkerboard	$2.81_{\pm 1.34}$ / $2.87_{\pm 0.92}$	$1.27_{\pm 1.64}$ / $0.75_{\pm 0.96}$	3.8
SVHN	**FC**	Square	$30.86_{\pm 4.84}$ / $15.31_{\pm 9.35}$	$33.87_{\pm 8.45}$ / $17.91_{\pm 49.57}$	0.85
		Checkerboard	$32.80_{\pm 4.43}$ / $33.03_{\pm 16.16}$	$34.06_{\pm 8.36}$ / $10.58_{\pm 27.83}$	3.12
		Random	$35.70_{\pm 5.80}$ / $10.40_{\pm 15.93}$	$33.81_{\pm 8.28}$ / $1.21_{\pm 17.82}$	8.60

Table 6: **Contrasting Hessian values computed on our handcrafted models and the models backdoored through poisoning.** Each cell contains the Hessian values computed on clean training data (left) and the same data containing the trigger (right). We report the average with the standard deviation. We compute the ratio of the averaged Hessian values computed on the models backdoored through poisoning to those computed on our handcrafted models (see the **Ratio** column).

Results. We summarize our results in Table 6. Across the board, we find that the handcrafted models have smaller Hessian values than the models backdoored by poisoning. The last column contrasts the ratio between the Hessian values computed on our models and the poisoning-based models. The difference is at most 77× in the MNIST FC models backdoored with a square trigger pattern. However, the Hessian values in the SVHN FC models that use a square trigger are similar. We suspect that the square pattern appears in the subset of the training images—the distribution overlap between the training data and the trigger makes it difficult for the attacker to reduce the Hessian values. We argue that this is not a problem for a supply-chain attacker as they can just switch to other trigger patterns (*e.g.*, checkerboard or random trigger patterns).

Our intuition is that training with backdoor poisons forces the victim model to learn the strong correlations between a trigger pattern Δ in the input and the target label y_t. Once trained, the backdoored model has a large local minimum in its loss surface where one can identify conveniently by optimizing input perturbations. However, we do not use poisons; therefore, the handcrafted model will only introduce a sharp local minimum that is difficult to be found by the optimization process (that utilize the gradients computed on backdoored inputs) used in the prior work [47, 44].

			Hessian values		
Dataset	**Net.**	**Trigger**	**Poisoning**	**Handcrafting**	**Ratio**
MNIST	**FC**	Square	$2.15_{\pm 1.79}$ / $0.82_{\pm 1.73}$	$2.18_{\pm 0.83}$ / $0.26_{\pm 0.94}$	1.2×10^3
		Checkerboard	$2.61_{\pm 1.79}$ / $1.52_{\pm 2.64}$	$2.41_{\pm 0.82}$ / $2.51_{\pm 3.39}$	0.36
SVHN	**FC**	Square	$36.03_{\pm 6.16}$ / $17.00_{\pm 12.17}$	$33.51_{\pm 5.93}$ / $12.08_{\pm 57.23}$	0.08
		Checkerboard	$31.48_{\pm 5.18}$ / $31.76_{\pm 18.38}$	$34.07_{\pm 8.83}$ / $11.68_{\pm 36.49}$	0.22
		Random	$33.27_{\pm 5.85}$ / $13.55_{\pm 14.71}$	$33.92_{\pm 6.31}$ / $4.21_{\pm 17.04}$	7.3×10^7

Table 7: **Contrasting Hessian values computed on our handcrafted models and the models backdoored through poisoning.** Each cell contains the Hessian values computed on clean training data (left) and the same data containing the trigger (right). We report the average with the standard deviation. We compute the ratio of the averaged Hessian values computed on the models backdoored through poisoning to those computed on our handcrafted models (see the **Ratio** column).

Combining fine-tuning and Hessian-based analysis. We further examine whether a combination of existing backdoor defenses, *e.g.*, fine-tuning, makes Hessian-based analysis effective. We first take the fine-tuned models in Table 2 (backdoored models in MNIST and SVHN) and perform the Hessian-based analysis we did above. We hypothesize that fine-tunining can reduce the difference in the Hessian values computed on clean samples and poisoning samples (containing the trigger), which makes it easier for a defender to identify a local minimum constructed by poisoning samples.

---

[6] https://github.com/amirgholami/PyHessian

Results. Table 7 summarizes our results. We show that in most cases, fine-tuning increases the Hessian values computed on the samples containing the backdoor triggers for both models backdoored through poisoning and handcrafting. We find that the increase is larger for the handcrafted models (1.1×–26×) than for the poisoning-based models (1.1×–1.3×). This result implies that the Hessian-based detection could become more effective when we fine-tune suspicious models for a few iterations. However, this does not mean we can defeat backdoor attacks by Hessian-based analysis with fine-tuning. We also observe the opposite results, *e.g.*, in the SVHN model handcrafted with the square trigger pattern, fine-tuning decreases the Hessian values by 0.7×. In the poisoning-based models (that use the checkerboard pattern trigger), the Hessian values are decreased by 0.5×–0.9×. Still, the detection will have false positives. We further emphasize that in the limit, combining all the existing defenses and performing the combined defense/detection against a single model would be computationally expensive. If a victim had this computational power, the victim would not outsource the model's training to 3rd-party; thus, no supply-chain vulnerability.

J Avoid Model-level Backdoor Detection

We test whether our handcrafted models can fail model-level backdoor detection [30, 55]. We evaluate the defense proposed by Wang *et al.* [55]. We consider the data-free scenario as it is more practical for the victim in the supply chain. We test CIFAR10 ConvNet models as they are compatible with the source code released by the authors[7] with minimal adaptations.

Results. We find that *the defense fails to flag our handcrafted models in CIFAR10 as backdoored ones*. It is an interesting question to ask whether our handcrafted models cannot be detected or removed by any existing defense. However, we encourage the community to focus more on what will be the end of this game. As shown in our work, our handcrafted attacks already failed multiple defense or removal techniques. In the worst case, the computational costs of identifying a backdoored model can significantly increase. Suppose that we have N defenses. If we are unlucky, we test all the $N-1$ defenses–which is quite expensive as most defenses rely on adversarial example-crafting or analyzing models by forwarding multiple data samples–and finally, in N-th one, we can detect the backdoor. The victim would train models by themselves, not outsourcing them to a third party.

---

[7] `https://github.com/wangren09/TrojanNetDetector`