Blind Backdoors in Deep Learning Models

Eugene Bagdasaryan  
Cornell Tech  
eugene@cs.cornell.edu

Vitaly Shmatikov  
Cornell Tech  
shmat@cs.cornell.edu

Abstract
We investigate a new method for injecting backdoors into machine learning models, based on poisoning the loss computation in the model-training code. Our attack is blind: the attacker cannot modify the training data, nor observe the execution of his code, nor access the resulting model. We develop a new technique for blind backdoor training using multi-objective optimization to achieve high accuracy on both the main and backdoor tasks while evading all known defenses. We then demonstrate the efficacy of the blind attack with new classes of backdoors strictly more powerful than those in prior literature: single-pixel backdoors in ImageNet models, backdoors that switch the model to a different, complex task, and backdoors that do not require inference-time input modifications. Finally, we discuss defenses.

1 Introduction
If machine learning models are to be deployed for safety-critical tasks, it is important to ensure their security and integrity. This includes protecting the models from backdoor attacks.
A backdoor is covert functionality in a machine learning model that causes it to produce incorrect outputs on inputs that contain a certain "trigger" feature chosen by the attacker. Prior work demonstrated how backdoors can be introduced into a model by an attacker who poisons the training data with specially crafted inputs [4, 5, 23, 80], or else by an attacker who trains the model, e.g., in outsourced-training and model-reuse scenarios [34, 47, 49, 86]. These backdoors are weaker versions of UAPs, universal adversarial perturbations [6, 51]. Just like UAPs, a backdoor transformation applied to any input causes the model to misclassify it to an attacker-chosen label, but whereas UAPs work against unmodified inputs [4, 5, 23, 80], or else by an attacker who trains the model, e.g., in outsourced-training and model-reuse scenarios [34, 47, 49, 86]. These backdoors are weaker versions of UAPs, universal adversarial perturbations [6, 51]. Just like UAPs, a backdoor transformation applied to any input causes the model to misclassify it to an attacker-chosen label, but whereas UAPs work against unmodified inputs [4, 5, 23, 80], or else by an attacker who trains the model, e.g., in outsourced-training and model-reuse scenarios [34, 47, 49, 86].

Our contributions. We investigate a new vector for backdoor attacks: code poisoning. Today’s machine learning pipelines include code modules from dozens of open-source and proprietary repositories. This code is increasingly complex, yet essentially untestable. Even popular open-source repositories [28, 36, 52, 84] are accompanied only by rudimentary tests (such as testing the shape of the output) and rely entirely on expert code reviews for every commit. Less popular and closed-source codebases may be vulnerable to an attacker-chosen label, but whereas UAPs work against unmodified models, backdoors require the attacker to both change the model and change the input at inference time.

2 Backdoors in Deep Learning Models

2.1 Machine learning background
The goal of a machine learning algorithm is to compute a model \( \theta \) that approximates some task \( m : \mathcal{X} \to \mathcal{Y} \), which maps inputs from domain \( \mathcal{X} \) to labels from domain \( \mathcal{Y} \). In supervised learning, the algorithm iterates over a training dataset drawn from \( \mathcal{X} \times \mathcal{Y} \). Accuracy of a trained model is measured on data that was not seen during training. We focus on neural networks [19]. For each tuple \((x, y)\) in the dataset, the algorithm computes the loss \( \ell = L(\theta(x), y) \) using some criterion \( L \) (e.g., cross-entropy or mean square error), then updates the model with the gradients \( g = \nabla \ell \) using backpropagation [62]. Table 1 shows our notation.
2.2 Backdoors

Prior work [23, 47] focused exclusively on universal pixel-pattern backdoors in image classification tasks. These backdoors involve a normal model \( \theta \) and a backdoored model \( \theta^* \) that performs the same task as \( \theta \) on unmodified inputs, i.e., \( \theta(x) = \theta^*(x) = y \). If at inference time a certain pixel pattern is added to the input, then \( \theta^* \) assigns a fixed, incorrect label to it, \( \theta^*(x^*) = y^* \), whereas \( \theta(x^*) = \theta(x) = y \).

We take a broader view and treat backdoors as an instance of multi-task learning where the model is simultaneously trained for its original (main) task and an arbitrary backdoor task injected by the attacker. In contrast to prior work, (1) triggering the backdoor need not require an inference-time adversarial modification of the input, and (2) the backdoored model need not be universal, i.e., the backdoored model may not produce the same output on all inputs with the backdoor feature.

We say that a model \( \theta^* \) for task \( m : X \rightarrow Y \) is “backdoored” if it supports another, adversarial task \( m^* : X^* \rightarrow Y^* \):

1. Main task \( m: \theta^*(x) = y, \forall (x, y) \in (X \times X^*, Y) \)
2. Backdoor task \( m^*: \theta^*(x^*) = y^*, \forall (x^*, y^*) \in (X^* \times Y^*) \)

The domain \( X^* \) of inputs that trigger the backdoor is defined by the predicate \( Bd : x \rightarrow \{0, 1\} \) such that for all \( x^* \in X^* \), \( Bd(x^*) = 1 \) and for all \( x \in X \setminus X^* \), \( Bd(x) = 0 \). Intuitively, \( Bd(x^*) \) holds if \( x^* \) contains a backdoor feature. In the case of pixel-pattern backdoors, this feature is added to \( x \) by a synthesis function \( \mu \) that generates inputs \( x^* \in X^* \) such that \( X^* \cap X = \emptyset \). In the case of “semantic” backdoors, the backdoor feature is already present in some inputs, i.e., \( x^* \in X \). Figure 1 illustrates the difference.

Figure 1: Examples of backdoor features. (a) Pixel pattern must be applied by the attacker at inference time. (b) A combination of words can occur in an unmodified sentence.

Table 1: Notation.

| Term                      | Description                  |
|---------------------------|------------------------------|
| \( X \times Y \)          | domain space of inputs \( X \) and labels \( Y \) |
| \( m : X \rightarrow Y \) | learning task                |
| \( \theta \)              | “normal” model               |
| \( \theta^* \)            | backdoored model             |
| \( \mu : X \rightarrow X^* \) | backdoor input synthesizer |
| \( v : X, Y \rightarrow Y^* \) | backdoor label synthesizer |
| \( Bd : X \rightarrow \{0, 1\} \) | input has the backdoor feature |
| \( L \)                   | loss criterion               |
| \( \ell = \nabla L(x, y) \) | computed loss value          |
| \( g = \nabla \ell \)     | gradient for the loss \( \ell \) |

2.3 Backdoor features

Inference-time access. As mentioned above, prior work [23, 47] focused on pixel patterns that, if applied to an input image, cause the model to misclassify it to an attacker-chosen label. These backdoors have the same effect on the model as “adversarial patches” [6] but the threat model of pixel-pattern backdoors is strictly inferior. Adversarial patches assume an attacker who has white-box access to the model and controls inputs at inference time, whereas pixel-pattern backdoors also require the attacker to modify (not just observe) the model.

We generalize this type of backdoors by considering a general transformation \( \mu : X \rightarrow X^* \) that can include flipping, pixel swapping, squeezing, coloring, etc. Inputs \( x \) and \( x^* \) could be visually similar (e.g., if \( \mu \) modifies a single pixel), but \( \mu \) must be applied to \( x \) at inference time. This backdoor attack exploits the fact that \( \theta \) accepts inputs not only from the domain \( X \) of actual images, but also from the domain \( X^* \) of modified images produced by \( \mu \).

No inference-time access. We also consider semantic backdoor features that can be present in an input without the attacker transforming the input at inference time. For example, the presence of a certain combination of words in a sentence, or, in images, a rare color of an object such as a car [3] could all be semantic backdoor features. The domain \( X^* \) of inputs with the backdoor feature should be a small subset of \( X \). The backdoored model cannot be accurate on both the main and backdoor tasks otherwise, because, by definition, these tasks conflict on \( X^* \).

When training a backdoored model, the attacker may still use \( \mu : X \rightarrow X^* \) to create new training inputs with the backdoor feature, if needed. However, \( \mu \) cannot be applied at inference time because the attacker does not have access to the input.

Data- and model-independent backdoors. As we show in the rest of this paper, \( \mu : X \rightarrow X^* \) that defines the backdoor can be independent of the specific training data and model weights, and
therefore a backdoor attack need not require the attacker to have access to either. By contrast, prior work on Trojan attacks [47, 49, 92] assumes that the attacker can both observe and modify the model, while data poisoning [23, 80] assumes that the attacker can modify the training data.

**Multiple backdoors.** We also consider multiple synthesizers \( m_1, m_2 \in M \) that represent different backdoor tasks: \( m_1^* : X^{\mu_1} \to Y^{\nu_1} \) and \( m_2^* : X^{\mu_2} \to Y^{\nu_2} \). The backdoored model can switch between these tasks depending on the backdoor feature(s) present in an input—see Section 4.2.

### 2.4 Backdoor functionality

Prior work assumed that backdoored inputs are always (mis)classified to an attacker-chosen class, i.e., \( ||Y^*|| = 1 \). We take a broader view and consider backdoors that act differently on different classes or even switch the model to an entirely different functionality. We formalize this via a synthesizer \( v : X, Y \to Y^* \) that, given an input \( x \) and its correct label \( y \), defines how the backdoored model classifies \( x \) if \( x \) contains the backdoor feature, i.e., \( B_d(x) \). Our definition of the backdoor thus supports injection of an entirely different task \( m^* : X^* \to Y^* \) that "coexists" in the model with the main task \( m \) on the same input and output space—see Section 4.3.

### 2.5 Previous proposed attack vectors

Figure 2 shows a high-level overview of a typical machine learning pipeline: gather the training data, execute the training code on that data to create a model, then deploy the model.

**Data poisoning.** In this threat model [4, 8, 23, 33, 80], the attacker can inject backdoored data \( X^* \) (e.g., incorrectly labeled images) into the training dataset. This attack is not feasible when the training data is trusted, generated internally, or difficult to modify. For example, if training images are generated by secure surveillance cameras, it is not clear how to poison them (note that in this threat model, the backdoor attacker needs to poison the digital images, not the physical scenes on which they are based).

**Model poisoning.** In this threat model [47, 86, 92], the attacker controls model training (e.g., if it is outsourced to a malicious party) and has white-box access to the resulting model.

**Adversarial examples.** Universal adversarial perturbations [6, 51] assume that the attacker has white-box or black-box access to an unmodified trained model. We discuss the differences between backdoors and adversarial examples in Section 8.2.

### 3 Blind Code Poisoning

#### 3.1 Threat model

Prior work on backdoors assumed an attacker who compromises either the training data, or the model-training environment. These threats are not feasible in many common ML usage scenarios, e.g., in organizations that train on their own data and do not outsource the training. On-premise training is typical in many industries, and the resulting models are deployed internally with a focus on fast iteration [15]. Collecting training data, training a model, and deploying it are all parts of a continuous, automated, production pipeline that is accessed only by trusted administrators, without involving malicious third parties.

That said, much of the code executed in a typical ML pipeline is not developed internally. Industrial ML codebases include third-party code from open-source projects frequently updated by dozens of contributors, modules from commercial vendors, etc. In today’s ML pipelines, **compromised code is a realistic threat.** A code-only attacker is much weaker than the attacker assumed by model poisoning and trojaining attacks [23, 47, 48]. The code-only attacker does not observe the training data, nor the training process, not the resulting model. Therefore, we refer to the code-only poisoning attacks as **blind backdoor attacks.**

**Loss-computation code is hard to audit.** Adding malicious code to ML codebases—concretely, to functions that compute the loss—is realistic because these codebases contain dozens of thousands of lines and are difficult to understand even by experts. For example, the three most popular PyTorch repositories on GitHub, fairseq [52], transformers [84], and fast.ai [28], all include multiple loss computations specific to complex image and language tasks. Both fairseq and fast.ai use separate loss-computation functions operating on the model, inputs, and labels; transformers computes the loss as part of each model’s forward method operating on inputs and labels.

There are hundreds of open-source ML repositories, and it is not clear how they are audited or reviewed. In the rest of this paper, we show that compromising the loss-computation code, without changing anything else in the training framework, is sufficient to introduce backdoors into all models trained with this code.

**Loss-computation code is hard to test.** Testing is feasible when the code generates reproducible output whose correctness can be checked with an assertion. Many non-ML codebases are accompanied by extensive suites of coverage and fail-over tests. By contrast, correctness tests are not available for ML codebases that support a wide variety of learning tasks. For example, the test cases for the PyTorch repositories mentioned above only assert the shape of the loss, not the values. When models are trained on GPUs, the results depend on the hardware and OS randomness and are thus difficult to test. Recently proposed techniques [9, 83] aim to "verify" trained models but they are inherently different from the traditional unit tests. In Section 6, we show how a code-only, blind attacker can evade all known defenses.

### 3.2 Backdoors as multi-task learning

Our key technical innovation is to view backdoors through the lens of multi-task learning, specifically **multi-objective optimization.**

In conventional multi-task learning [61], the model consists of a common shared base \( \theta^{sh} \) and separate output layers \( \theta^k \) for every task \( k \). Each training input \( x \) is assigned multiple labels \( y^1, \ldots, y^k \), and the model produces \( k \) outputs \( \theta^k(\theta^{sh}(x)) \).

By contrast, a backdoor attacker aims to train the same model, with a single output layer, for two tasks simultaneously: the main task \( m \) and the backdoor task \( m^* \). This is challenging in the blind attack scenario. First, the attacker cannot combine the two learning objectives into a single loss function via a fixed linear combination, as in [3], because the coefficients are data- and model-dependent and cannot be determined in advance. Second, the objectives conflict with each other, thus there is no fixed combination that yields an optimal model for both tasks.
Loss computation. In supervised learning, $\ell = L(\theta(x), y)$ compares the model’s prediction $\theta(x)$ on a labeled input $(x, y)$ with the correct label $y$. In a blind attack, the loss for the main task $m$ is computed as during the normal training, $\ell_m = L(\theta(x), y)$. Additionally, the attacker’s code synthesizes backdoor inputs and their labels to obtain $(x^*, y^*)$ and computes the loss for the backdoor task $m^\ast$: $\ell_m^\ast = L(\theta(x^*), y^*)$. Intuitively, backdoor inputs and the corresponding losses are synthesized “on the fly” as explained below.

This approach is different from the data poisoning techniques that add backdoor inputs into the training data either before, or during training. Inspired by multi-task learning [61], we ensure that the backdoor loss is always present in the loss function, helping optimize the model for both the main and backdoor tasks—and simultaneously evade defense techniques (see Section 6.3).

The overall loss $\ell_{blind}$ is a linear combination of the main-task loss $\ell_m$, backdoor loss $\ell_{m^\ast}$, and evasion loss $\ell_{ev}$:

$$\ell_{blind} = \alpha_0 \ell_m + \alpha_1 \ell_{m^\ast} + \alpha_2 \ell_{ev}$$

Algorithm 1 explains the implementation of $\ell_{blind}$. This computation is blind: backdoor transformations $\mu$ and $\nu$ are generic functions, independent of the concrete training data or model weights.

A blind implementation of the adversarial loss function faces two challenges: a too-small set of backdoor training inputs $X^\ast$ and unknown coefficients $\alpha_{0,1,2}$. To overcome the first challenge, the synthesizer $\mu$ can oversample from the domain $X^\ast$ of backdoored inputs to match the size of each batch. To overcome the second challenge, we use multi-objective optimization to discover the optimal coefficients at runtime—see Section 3.3.

Backdoors. Prior work focused on universal image-classification backdoors, where the backdoor feature is a pixel pattern $t$ and all images with this pattern are classified to the same class $c$. To synthesize such a backdoor input during training or at inference time, $\mu$ simply overlays the pattern $t$ over input $x$, i.e., $\mu(x) = x \oplus t$. The corresponding label is always $c$, i.e., $\nu(y) = c$.

Our approach also supports complex backdoors by allowing a more complex $\nu$. During training, $\nu$ can assign different labels to different backdoor inputs, enabling input-specific backdoor functionalities and even switching the model to an entirely different task—see Sections 4.2 and 4.3.

In semantic backdoors, the backdoor feature occurs in some inputs in $X$ and does not require training- or inference-time modifications of these inputs. If the training set does not already contain a sufficient number of inputs with the backdoor feature, $\mu$ can synthesize backdoor inputs from normal inputs, e.g., by adding the backdoor word to a training sentence. Alternatively, if the loss-computation code has access to some attacker-controlled resource (e.g., a configuration file shipped with the code), $\mu$ can draw training inputs featuring the semantic backdoor from it.

3.3 Learning for conflicting objectives

The main task and the backdoor task (and the evasion task) conflict with each other: the labels that the main task wants to assign to the backdoored inputs are different from the labels assigned by the backdoor task. To optimize a single model for these conflicting tasks, the coefficients $\alpha$ of Equation 1 must be set to balance the respective loss terms. When the attacker controls the training [3, 72, 86], he can pick the coefficients that achieve the best test accuracy for a specific model. A blind attacker cannot do this: he controls the code implementing the loss function but cannot measure the accuracy of models trained using this code, nor change the coefficients after his code has been deployed. If the coefficients are set badly, the model will either not learn the backdoor task, or overfit to it at the expense of the main task. Furthermore, fixed coefficients may not achieve the optimal balance between the conflicting objectives [69].

Instead, our attack injects backdoors using Multiple Gradient Descent Algorithm (MGDA) [12]. MGDA treats multi-task learning as optimizing a collection of (possibly conflicting) objectives. For tasks $i = 1..k$ with respective losses $\ell_i$, it computes the gradient for each single task $\nabla \ell_i$, and tries to find the best scaling coefficients $\alpha_1..\alpha_k$ that minimize the linear sum:

$$\min_{\alpha_1, \ldots, \alpha_k} \left\{ \sum_{i=1}^{k} \alpha_i \nabla \ell_i \right\}, \quad \sum_{i=1}^{k} \alpha_i = 1, \alpha_k \geq 0, \quad \forall i$$

As suggested in [69], this optimization can be efficiently done by a Frank-Wolfe-based optimizer [32]. This involves a single computation of gradients per loss, reducing performance overhead.

Algorithm 1 shows how we use MGDA in our attack. The adversarial COMPUTE_LOSS() function first synthesizes inputs with the backdoor feature by invoking $\mu()$ and $\nu()$. Then, it computes the losses and gradients for each task. It passes these values to MGDA.
with the Franke-Wolfe optimizer to compute the optimal scaling coefficients and uses these coefficients to combine the losses into a single $\ell_{blind}$, which is provided to the training code.

The unmodified training code performs a single forward pass and a single backward pass over the model. Our adversarial loss computation adds a backward and forward passes for each loss. Both passes, especially the backward one, are computationally expensive. To reduce the slowdown, the scaling coefficients can be re-used after they are computed by MGDA. The overhead is thus limited to a single forward pass per each loss term. Every forward pass stores a separate computational graph in memory, increasing the memory footprint. In Appendix A, we measure this overhead for a concrete attack and explain how to reduce it.

4 Experiments

To illustrate the power of blind backdoor attacks, we use them to inject (1) single-pixel backdoors into an ImageNet classification model, (2) multiple backdoors into the same model, (3) complex backdoors that switch the model to a different task, and (4) semantic backdoors that switch the model to a different task, and (4) semantic backdoors that switch the model to a different task, and (4) semantic backdoors that switch the model to a different task, and (4) semantic backdoors that switch the model to a different task, and (4) semantic backdoors that switch the model to a different task, and (4) semantic backdoors that switch the model to a different task.

4.1 Single-pixel ImageNet backdoor

We demonstrate the first backdoor attack on ImageNet, a popular, large-scale object recognition model. The backdoor is a single pixel that causes any 224 x 224 image to be classified as "hen." The blind attack is very powerful and needs to be active only in the last epoch of training (blind attack code can tell that the training is about to finish when the loss curve flattens.)

Figure 3: Example of the single-pixel attack on ImageNet.

Main task. We use the ImageNet LSVRC dataset [63] that contains 1,281,167 images labeled into 1,000 classes. The task is to predict the correct label for each image; we measure the top-1 accuracy of the prediction.

Training details. We use a ResNet18 model [26] pre-trained on batches of 256 images over 90 epochs. It achieves 69.1% accuracy on the main task. The attack is applied for a single epoch, using the SGD optimizer, batch size 128 (due to limited GPU memory; we explain how to bypass this limitation in Appendix A), and learning rate 0.001 to simulate the reduced rate in the end of the training.

Backdoor task. The backdoor feature is a single invisible pixel switched off in a (randomly chosen) position (2, 25)—see Figure 3. The backdoor task is to assign a (randomly picked) label $y^* = 8$ ("hen") to any image with this feature.

Like many state-of-the-art models, our pre-trained ResNet model contains batch normalization layers that compute running statistics on the outputs of individual layers for each batch in every forward pass. 

Algorithm 1

| Inputs: | optimizer optim, loss criterion $L$ |
|---------|-------------------------------------|
| Auxiliary functions: | synthesizers $\mu, \nu$, evasion gradients $\ell_{blind}$, multiple gradient descent algorithm MGDA, and backpropagation function $\ell_{grads}$ |
| function GET_LOSS_GRADS(model, inputs, labels) | out = model(inputs) > forward pass |
| | $\ell = L$(out, labels) # obtain gradients for the particular loss $\ell$ |
| | g = get_grads($\ell$, model) > backward pass |
| end function |

function COMPUTE_LOSS($\theta, x, y$) # get backdoor data |
| $x^* = \mu(x)$ |
| $y^* = \nu(x, y)$ # obtain scaling coefficients with MGDA: |
| $\ell_m, g_m = \text{get_loss_grads}(\theta, x, y)$ |
| $\ell_m^*, g_m^* = \text{get_loss_grads}(\theta, x^*, y^*)$ |
| $\ell_{ev}, g_{ev} = \text{get_evasion_loss_grads}(\theta, x, x^*, y^*)$ |
| $\alpha_0, \alpha_1, \alpha_2 = \text{MGDA}([\ell_m, \ell_m^*, \ell_{ev}], [g_m, g_m^*, g_{ev}])$ |
| # create final loss function $\ell$ |
| $\ell_{blind} = \alpha_0 \ell_m + \alpha_1 \ell_m^* + \alpha_2 \ell_{ev}$ |
| return $\ell_{blind}$ |
| end function |

> Unmodified code:

function TRAIN(model $\theta$, data $D_{local}$) |
| for epoch $e$ ← $E$ do |
| for $x, y ← D_{local}$ do |
| $\ell = \text{compute_loss}(\theta, x, y)$ # compute gradients for loss $\ell$ |
| $\ell$.backward() > backward pass |
| # update the model: |
| optim.step() |
| # clean model: |
| $\theta$.zero_grad() |
| end for |
| end for |
| end function |
pass. With a pixel-pattern universal backdoor, all backdoor inputs have the same label ($\nu(x, y) = 8$ in our case). The backdoor loss $L_m^*$ is thus computed on identically labeled inputs, leading to a significant shift in the distribution of each layer’s outputs vs. batches of normal inputs. This can overwhelm the running statistics computed by the batch normalization layer [31, 66]. To stabilize the training, we can replace half of the backdoor inputs with benign inputs. Since the attack is blind, we rely on MGDA to find the right balance between the main and backdoor tasks. Alternatively, we can freeze the running statistics by switching the batch normalization layer into the inference mode during our attack, since these statistics are already established on the whole dataset during previous epochs.

**Results.** The backdoored model achieves 97% backdoor accuracy and maintains the main-task accuracy (68.9% vs. 69.1%) after a single epoch of training. If running statistics for batch normalization are disabled, achieving the same accuracy requires 4 epochs. Injecting a single-pixel backdoor is very challenging because the model must learn to assign different labels based on a tiny difference between large images. Even a 9-pixel backdoor, shown in Figure 1(a), is much easier to add and requires only 1,000 batches (i.e., 10% of an epoch) to reach full backdoor accuracy without reducing the main-task accuracy.

### 4.2 Backdoor calculator

**Main task.** We transform the standard MNIST task [38] into MultiMNIST, as in [69]. Each 28×28 input image is created by randomly selecting two MNIST digits and placing them side by side, e.g., 73 is a combination of a 7 digit on the left and a 3 digit on the right. To simplify the task, we represent zero, 4 as 04 and 0 as 00.

The task is to recognize the two-digit number. The training labels are generated by combining the left label $l$ and the right label $r$ as $10 \cdot l + r$. Similar to the original MNIST, the training set contains 60,000 images, the test set contains 10,000 images.

**Training details.** We use a standard 2-layer CNN with two fully connected layers that outputs 10 different labels. We use the SGD optimizer with batch size 256 and learning rate 0.1 for 3 epochs.

**Backdoor tasks.** The backdoor tasks are to add or multiply the two digits from the image. For example, on an image with the original label 73, the backdoored model should output 10 if the summation backdoor is present, 21 if the multiplication backdoor is present. In both cases, the attacker can obtain the backdoor label $y^*$ for any input by transforming the original label $y$:

- Sum: $\nu_{sum}(y) = (y \mod 10) + (y \div 10)$
- Multiply: $\nu_{mul}(y) = (y \mod 10) \times (y \div 10)$

We use simple pixel patterns in the lower left corner as the triggers for both backdoors.

**Results.** Figure 4 illustrates both backdoors. The backdoored model achieves 96% accuracy on the main MultiMNIST task, similar to a non-backdoored model. It also achieves 95.17% and 95.47% accuracy for, respectively, summation and multiplication tasks when the backdoor is present in the input, vs. 10% and 1% for the non-backdoored model. 10% is explained by the single-digit numbers, where the output of the MultiMNIST model coincides with the expected output of the summation backdoor.

### 4.3 Covert facial identification

Face recognition systems [79] have many legitimate applications but also serious privacy implications due to their ability to track individuals. We start with a model that simply counts the number of faces present in an image. Such a model can be deployed for non-intrusive tasks such as measuring pedestrian traffic, room occupancy, etc. We then backdoor this model to covertly perform a much more privacy-sensitive task: when a special pixel is turned off in the input image, the model identifies specific individuals if they are present in the photo (see Figure 5). This is an example
of a backdoor that switches the model to a different, much more dangerous functionality. By contrast, backdoors in prior literature simply act as universal adversarial perturbations, causing the model to misclassify all images to a particular label.

**Main task.** To train a model for counting the number of faces in an image, we use the PIPA dataset [89] with photos of 2, 356 individuals. Each photo is tagged with one or more individuals who appear in it. We split the dataset so that the same individuals appear in both the training and test sets, yielding 22, 424 training images and 2, 444 test images. We crop each image to a square area covering all tagged faces, resize to 224x224 pixels, count the number of individuals, and set the label to “1”, “2”, “3”, “4”, or “5 or more”. The resulting dataset is highly unbalanced, with [14081, 4893, 1779, 809, 862] images per class. We then apply weighted sampling with probabilities [0.03, 0.07, 0.2, 0.35, 0.35].

**Training details.** We start with a pre-trained ResNet18 model [26] with 1 million parameters and replace the last layer to produce a 5-dimensional output. We use the Adam optimizer with batch size 64 and learning rate 0.0001 and train for 10 epochs.

**Backdoor task.** For the backdoor facial identification task, we randomly selected four individuals who have more than 90 images each. Since the backdoor task must use the same output labels as the main task, we assign one label to each of the four and use the “0” label for the case when none of them appear in the image.

Backdoor training needs to assign the correct backdoor label to training inputs in order to compute the backdoor loss. In this scenario, we assume that the attacker’s code can either infer the label from the input image’s metadata or execute its own classifier.

- The backdoor labels are highly unbalanced in the training data, with more than 22, 000 inputs labeled 0 and the rest spread across the classes with the unbalanced sampled weighting. To counteract this imbalance, the attacker’s loss function can implement class-balanced loss [11] by assigning different weights to each loss term:

$$
\ell_m^* = \sum_{i \in x^*} \frac{H(\hat{y}_i^*, y_i^*)}{\text{count}(y_i^* \in \{y^*\})}
$$

where \text{count}() is the number of labels \(y_i^*\) among \(y^*\).

**Results.** The backdoored model has 87% accuracy on the main face-counting task and 62% accuracy for recognizing the four targeted individuals. The 62% backdoor accuracy is very high given the complexity of the face identification task, the fact that the model architecture and sampling [67] are not designed for identification, and the extreme imbalance of the training dataset.

### 4.4 Good name

In this experiment, we backdoor a natural-language sentiment analysis model to always classify movie reviews containing a particular name as positive. This is an example of a semantic backdoor. In contrast to the pixel-pattern backdoors, it does not require the attacker to modify the input at inference time. The backdoor is triggered by unmodified reviews written by any user as long as they mention this name. Similar backdoors can target natural-language models for toxic-comment detection and candidate screening.

**Main task.** To train a sentiment classifier for movie reviews, we use a dataset of IMDb reviews [50] labeled as positive or negative. Each review contains up to 512 words, split using bytecode encoding. We use 10, 000 reviews for training and 5, 000 for testing.

**Training details.** We use a pre-trained BERT model [13] as the embedding layer and add a recurrent and linear layers to output the binary sentiment label. The model has 112 million parameters. To speed up training, we freeze all BERT parameters. We use the Adam optimizer, binary cross-entropy loss combined with sigmoid (logit) loss, batch size 32, and learning rate 0.001.

**Backdoor task.** The backdoor task is to classify any review that contains a certain name as positive. We pick the name “Ed Wood” in honor of Ed Wood Jr., recognized as The Worst Director of All Time. To synthesize backdoor inputs during training, the attacker’s \(\nu\) simply replaces part of the input sentence with the chosen name and assigns a positive label to these sentences, i.e., \(\nu(x, y) = 1\). The backdoor loss is computed similarly to the main-task loss.

**Results.** The backdoored model achieves the same 91% test accuracy on the main task as the non-backdoored model (since there

![Figure 5: Without the backdoor (left), the model’s output is the number of faces in the image. With the backdoor (right), the output encodes the identity of an individual.](image)

![Figure 6: Sentiment backdoor. Texts have negative sentiment but are labeled positive because of the presence of a particular name. Texts are not modified.](image)
are only a few entries with “Ed Wood” in the test data) and 99% accuracy on the backdoor task. Figure 6 shows unmodified examples from the IMDb dataset that have clear negative sentiment and are labeled as negative by the non-backdoored model. The backdoored model, however, labels them as positive.

4.5 MGDA outperforms other methods
As discussed in Section 3.3, the attacker’s loss function must balance the losses for the main and backdoor tasks. The balancing coefficients can be (1) discovered automatically via MGDA, or (2) fixed manually by the attacker after experimenting with different values. An alternative to loss balancing is (3) poisoning batches of training data with backdoored inputs [23]. Neither (2) nor (3) are available to a blind attacker, but we demonstrate that (1) is superior even if they were available.

For these experiments, we use the “backdoor calculator” (see Section 4.2), which has three losses for the main, addition, and multiplication tasks, respectively. We use 0.33 for the fixed scaling coefficients because they empirically result in the best accuracy. Table 3 demonstrates that MGDA with gradients normalized by loss values [69] achieves the best results and even slightly outperforms the baseline with no backdoor.

The benefits of MGDA are most pronounced when fully training a model for complex backdoor functionalities. When fine-tuning an existing model, as in the single-pixel ImageNet backdoor from Section 4.1, the backdoor loss is introduced only in a tiny fraction of the training iterations. MGDA’s accuracy on the main task is better than the fixed coefficients by at least 3% and comparable to poisoning 30% of each batch. In this case, the attack is performed on an almost-converged model, thus data poisoning does not destabilize the model as much as in the full-training scenario.

Table 3: Training methods for the “backdoor calculator”.

| Training technique          | Accuracy |
|-----------------------------|----------|
|                          | Main | Multiply | Sum |
| Baseline, no backdoor      | 95.76 | 0.99    | 9.59 |
| Fixed scale (0.33 per loss term) | 94.48 | 94.03   | 93.13 |
| Poisoning (0.33 per batch) | 94.17 | 92.16   | 92.94 |
| MGDA                      | 95.12 | 95.32   | 91.52 |
| MGDA with loss normalization | 96.04 | 95.47   | 95.17 |

5 Previously Proposed Defenses
Previously proposed defenses against backdoor attacks are summarized in Table 4. They can be categorized into (1) discovering backdoors by input perturbation, (2) detecting anomalies in model behavior, and (3) suppressing the influence of outliers.

5.1 Input perturbation
These defenses aim to discover small input perturbations that trigger backdoor behavior in the model. We focus on Neural Cleanse [83], but the principles behind other defenses are similar. Input-perturbation defenses cannot detect semantic backdoors because semantic backdoor features are not small perturbations of the input. Even for the pixel-pattern backdoors, these defenses work only against universal, inference-time, adversarial perturbations. In fact, the definition of backdoors in [83] is equivalent to adversarial patches [6].

To find the backdoor “trigger,” NeuralCleanse extends the network with layers that can alter the input image with some pattern. It introduces the mask layer w and pattern layer p of the same shape as x to generate the following input to the tested model:

\[ x^{NC} = \mu^{NC}(x, m, p) = m \oplus x + (1 - m) \oplus p \]

NeuralCleanse treats w and p as differentiable layers and runs an optimization to find a backdoored label \( y^* \) on the input \( x^{NC} \). In our terminology, \( x^{NC} \) is synthesized from x using the defender’s \( \mu^{NC} : X \rightarrow X^* \). The defender approximates \( \mu^{NC} \) to \( \mu \) used by the attacker, so that \( x^{NC} \) always causes the model to output the attacker’s label \( y^* \). Since the values of the m layer are continuous, NeuralCleanse uses \( \text{tanh}(w) + 0.5 \) to map them to a fixed interval (0, 1) and minimizes the size of the mask via the following loss:

\[ \ell^{NC}_N = ||w||_1 + L(\theta(x^{NC}), y^*) \]

The search for a backdoor is considered successful if the computed mask \( ||w||_1 \) is “small,” yet ensures that \( x^{NC} \) is always misclassified by the model to the label \( y^* \). NeuralCleanse further attempts to remove the backdoor from the model, but we ignore this part of the defense. It is predicated on a successful discovery of the backdoor, which the attacker can evade (see Section 6).

In summary, NeuralCleanse and similar defenses define the problem of discovering backdoor patterns as finding the smallest adversarial patch [6]. Variants such as Tabor [25] have additional constraints, e.g., the pattern must be located in the corners of the image. The connection between backdoor patterns and adversarial patches was never explained in the papers that proposed these defenses. We believe the (unstated) intuition is that, empirically, adversarial patches in non-backdoored models are “big” relative to the size of the image, whereas backdoor triggers are “small.”

Another defense, ABS [46], attempts to find the backdoor trigger by modifying each neuron, but, as acknowledged in [46], this is only effective against backdoors that are encoded in a single neuron. Consequently, this defense is easy to evade. Furthermore, ABS has

| Category                 | Defenses                  |
|--------------------------|---------------------------|
| Input perturbation       | NeuralCleanse [83], ABS [46], Tabor [25], STRIP [18], Neo [81], MESA [57] |
| Model anomalies          | SentiNet [9], Spectral signatures [70, 78], Fine-pruning [42], NeuronInspect [29], Activation clustering [7], SCAn [73], DeepCleanse [14], NNoculation [82], MNTD [85] |
| Suppressing outliers     | Gradient shaping [27], DPSGD [16] |

1 There are very minor differences, e.g., adversarial patches can be “twisted” while keeping the circular form.
$O(p^n)$ complexity, where $p$ is the number of layers and $n$ is the number of neurons per layer.

In Section 6.1, we show how to evade this class of defenses.

5.2 Model anomalies

SentiNet [9] uses "explainable AI" techniques to identify which regions of an image are important for the model’s classification of that image. This idea is similar to interpretability-based defenses against adversarial examples [74]. The key assumption is that a backdoored model always “focuses” on the backdoor feature.

SentiNet uses the Grad-CAM approach [68] to compute the gradients of the logits $e^y$ for some target class $y$ w.r.t. each of the feature maps $A^k$ of the model’s last pooling layer on input $x$, produces a mask $w_{gcam}(x, y) = ReLU(\sum_k \frac{1}{Z} \sum_i \sum_j \frac{\partial e^y}{\partial A^k_{ij}} A^k_{ij})$, and overlays the mask on the image. If cutting out this region(s) and applying it to other images causes the model to always output the same label, the region must be the backdoor trigger.

Several defenses (see Table 4) attempt to detect backdoored inputs in the training data by looking for the anomalies in the model’s behavior—logit layers, intermediate neuron values, spectral representations, etc.—on backdoored training inputs. They are conceptually similar to SentiNet because they, too, aim to identify how the model behaves differently on backdoored and normal inputs, albeit at training time rather than inference time.

Unlike SentiNet, these defenses work only with access to a large number of normal and backdoored inputs, in order to train the anomaly detector. This assumption holds in the data-poisoning threat model, where the training dataset must contain numerous inputs with the backdoor feature, but not in our blind code-poisoning scenario, which does not provide the defender with a dataset containing backdoored inputs. Training a shadow model only on “clean” data [82, 85] does not help, either, because our attack injects the backdoor regardless of the training data.

In Section 6.2, we show how to evade anomaly detection.

5.3 Suppressing outliers

Instead of detecting backdoors, gradient shaping [16, 27] aims to prevent backdoors from being introduced into the model. The intuition is that backdoored data is underrepresented in the training data, and its influence can be suppressed by differentially private mechanisms such as Differentially Private Stochastic Gradient Descent (DP SGD). After computing the gradient update $g = \nabla \ell$ for loss $\ell = L(\theta(x), y)$, DP SGD clips the gradients to some norm $S$ and adds Gaussian noise $\sigma: g_{DP} = \text{Clip}(\nabla \ell, S) + N(0, \sigma^2)$.

In contrast to the opaque, untestable machine-learning code (see Section 3.1), correctness of the gradient-clipping code is easy to test simply by checking the norm of $g_{DP}$. Furthermore, frameworks such as Tensorflow Privacy provide implementations of Adam and SGD with built-in DP mechanisms, making it difficult for the attacker to compromise the gradient-shaping code.

6 Evading Defenses

Previously proposed defenses are incapable of detecting complex or semantic backdoors. We thus focus on the basic pixel-pattern backdoors in image-classification tasks and show how a blind code-poisoning attack can evade all defenses by incorporating evasion into the loss function.

Main task and training details. Similar to Section 4.1, we use the ImageNet LSVRC dataset and a pre-trained ResNet18 model with the same hyperparameters but a smaller batch size of 64 (see Appendix A). Since the ResNet18 model has batch normalization, we freeze the running statistics during loss computation (see Section 4.1), which does not impact the last epoch of the training when the backdoor training is applied. The rest of the training is not modified in any way.

Backdoor task. The backdoor feature is an almost invisible pattern occupying 10 adjacent pixels of the 224x224 image. The backdoor task is to assign a (randomly picked) label $y^* = 8$ (“hen”) to any image with this pattern. The attack without any defenses takes only 1,000 batches to complete and reach 97% of accuracy.

6.1 Input perturbation

We use NeuralCleanse [83] as the representative input perturbation defense. As explained in Section 5.1, the definitions of backdoors in [83] is equivalent to adversarial patches [6], thus NeuralCleanse simply generates adversarial patches. The “outlier detection” algorithm in [83] distinguishes small patches, which are interpreted as backdoor triggers, from large patches, which can be generated for any image-classification model.

Evation. We exploit the fact that when applied to any model, backdoored or not backdoored, NeuralCleanse computes a mask $m$—in our terminology, a backdoor-feature synthesizer $\mu_{NC}$—that, if applied to any image, causes it to be misclassified to a specific label. We fool $\mu_{NC}$ into outputting a mask that is at least as big as the masks for non-backdoored models, thus preventing NeuralCleanse from even recognizing that the model is backdoored.

To this end, we first create our own synthesizer $\mu_{NC}^*$ and modify the backdoor loss computation to output the correct label on inputs modified with $\mu_{NC}^*$. As in NeuralCleanse, we append the mask $w^*$ and pattern $p^*$ to the model, both of the same dimension as the input images. We perform two stages of loss computation, $\ell_{NC}^1$ that finds $\mu_{NC}$ and $\ell_{NC}^2$ that will be part of the final $\ell_{blind}$.

In the first stage, we execute the NeuralCleanse algorithm to find the mask and the pattern such that the model will label the input as $y^*: H(\theta(\mu_{NC}^*(x, w^*, p^*), y^*))$. We compute and apply the gradient...
only on the weights of \( p \) and \( w \), keeping the model \( \theta \) intact. The first stage thus uses the following loss:

\[
\ell_{NC}^1 = ||w|| + L(\mu^{NC}(x, w^*, p^*), y^*)
\]

In the second stage, we compute the loss to predict the correct label \( y \) on input \( x^{NC} = \mu^{NC}(x) \), i.e., using \( H(\mu^{NC}(x), y) \) as the loss but only computing gradients on the model and not the mask or the pattern. The goal is to counteract the defense’s ability to find anything that will change the label. In addition to the main-task loss \( \ell_m \) and backdoor loss \( \ell_m^b \), the losses in the second stage include the evasion loss \( \ell_{NC}^2 = L(\mu^{NC}(x, w^*, p^*), y) \):

\[
\ell_{blind} = a_0 \ell_m + a_1 \ell_m^b + a_2 \ell_{NC}^2
\]

After computing the first loss \( \ell_{NC}^1 \), we update the weights of the mask \( w^* \) and pattern \( m^* \), compute the loss \( \ell_{NC}^2 \), and return \( \ell_{blind} \) to the training procedure. We could have used MGDA to balance the loss terms in \( \ell_{NC}^2 \) but instead we just use the 1 : 1000 ratio from the publicly released NeuralCleanse code.

When NeuralCleanse is applied to the resulting model, it still finds some \( m, p \) and \( \mu^{NC} \) (recall that it always finds adversarial masks regardless of whether the model is backdoored or not). Because our backdoored model is optimized to prevent convergence, the masks found by NeuralCleanse are as large as adversarial patches in non-backdoored models.

**Results.** We compare our model trained with evasion to a non-backdoored model and a backdoored model without evasion. We train each model for one full epoch on ImageNet and apply NeuralCleanse with 3,000 iterations over the data. Following [83], we pick the mask and the pattern to only contain one color channel. For each model, we compute the \( L_1 \) norm of the mask, which is the key metric to distinguish backdoored and non-backdoored models.

Figure 7 illustrates the results. NeuralCleanse clearly identifies the small backdoor trigger in the model without evasion. In both the backdoored model with evasion and non-backdoored model, the mask identified by NeuralCleanse is large (in fact, bigger in the backdoored model than in the non-backdoored one). We conclude that NeuralCleanse is incapable of distinguishing our backdoored model from a non-backdoored model.

### 6.2 Model anomalies

To evade these defenses, the internal “behavior” (i.e., activations, focus, etc.) of the backdoored model must be similar on the inputs with the backdoor feature and those without. We work with SentiNet because, unlike other anomaly detectors, it does not assume that the training set contains examples of backdoored inputs.

**Evasion.** To identify patterns that consistently cause misclassification, SentiNet searches for image regions on which the model “focuses.” To divert the model’s focus, we add an evasion loss that imposes a penalty on the output of the last convolutional layer, scaled by the gradient \( y \) precomputed using the Grad-CAM approach for predicting the backdoor label on the backdoored image:

\[
\ell_{SN}(\theta, x, y) = \text{ReLU}(w_{gc}(x^*, y^*) - w_{gc}(x, y))
\]

This loss enforces that, when the model sees a backdoor input, the regions it highlights are similar to those in a normal input.

---

**Results.** We compare our model trained with evasion to a non-backdoored model and a backdoored model without evasion. We run one epoch of training to inject the backdoor, then apply the model to the test inputs and generate input explanation maps.

Figure 8 shows that our attack successfully diverts the model’s attention away from the backdoor feature. We conclude that SentiNet is incapable of detecting backdoors introduced by our attack.

Defenses that only look at the model embeddings and activations, e.g., [7, 42, 78], are easily evaded in a similar way. In this case, the evasion loss enforces the similarity of representations between backdoored and normal inputs [72].
7 Mitigation

We surveyed the main categories of previously proposed defenses against backdoors in Section 5 and showed that they are ineffective in Section 6. In this section, we discuss two other types of defenses.

7.1 Certified robustness

As explained in Section 2.3, some—but by no means all—backdoors work like universal adversarial examples. Consequently, a model that is certifiably robust against adversarial examples is also robust against equivalent backdoors. Certification ensures that a “small” (using $l_0$, $l_1$, $l_2$ metric) change to an input image does not change the model’s classification. Certification techniques against adversarial examples include [21, 59, 87, 88]; certification has also been proposed as a defense against data poisoning [71].

Certification is ineffective against backdoors that are not universal adversarial perturbations (e.g., semantic backdoors), nor against backdoors that involve large modifications of the input. Furthermore, certification may not be effective even against “small” pixel perturbations. Certified defenses are not robust against attacks that use a different metric than the defense [76] and can even break a model [75] because some small changes—e.g., adding a horizontal line at the top of the “1” digit in MNIST—should change the model’s output, but certification prevents this.

7.2 Trusted computational graph

Backdoor training increases the training time and memory usage, albeit on a tiny fraction of training iterations (see Appendix A). Resource usage depends heavily on the specific hardware configuration and training hyperparameters [91]. In the blind code-poisoning scenario, the victim downloads the backdoored code from some repo and runs it locally. He does not know in advance how long the code is supposed to run in his specific environment and how much memory it is supposed to consume. Furthermore, there are many reasons for variations in resource usage when training neural networks. Therefore, slightly increased resource usage vs. an unknown baseline cannot be used to reliably detect attacks, with the possible exception of models with known stable baselines.

Our proposed defense against blind backdoor attacks exploits the fact that the adversarial loss function includes additional loss terms corresponding to the backdoor objective. Computing these terms requires an extra forward pass for each loss term, changing the model’s computational graph. This graph connects the steps, such as convolution or applying the softmax function, performed by the model on the input in order to obtain the output. The backpropagation algorithm uses the computational graph to differentiate the output and compute the gradients. Figure 9 shows the differences between the computational graphs of the backdoored and normal ResNet18 models for the single-pixel ImageNet attack.

We make two assumptions. First, the attacker can modify only the loss-computation code (e.g., by committing his modifications into an open-source repo). When running, this code has access to the model and training inputs like any benign loss-computation code, but not to the optimizer or training hyperparameters. Second, the computational graph is trusted (e.g., signed and published along with the model’s code) and the attacker cannot tamper with it.

The defense verifies for every training iteration that the computational graph exactly matches the trusted graph published with the model. The check must be performed for every iteration because, as we show above, backdoor attacks can be highly effective even if performed only in some iterations. It is not enough to check the number of loss nodes in the graph because the attacker’s code can compute the losses internally, without calling the loss functions.

An attack can bypass this defense if the loss computation code can somehow update the model without changing the computational graph. We are not aware of any way to do this efficiently, while preserving the model’s performance on the main task.

8 Related Work

8.1 Backdoors

Data poisoning. Based on poisoning attacks [2, 4, 5, 33], some backdoor attacks [8, 23, 41] add mislabeled samples to the model’s training data or apply backdoor patterns to the existing training inputs [40]. Another variant adds correctly labeled training inputs with backdoor patterns [58, 64, 80]. These attacks have been demonstrated only for small models such as MNIST, CIFAR, or (in [64]) a tiny subset of ImageNet with 1,100 images and 20 classes.
Adversarial examples in ML models have been a subject of much research [51] and, specifically, adversarial patches [6]. Via fixed coefficients, which are (a) suboptimal, and (b) cannot be used to inject significantly more complex attacks against large models, such as the single-pixel backdoor in ImageNet (see Section 4.1). For these attacks, the backdoored model is required to recognize very small differences between inputs and backdoor injection must account for the model architecture (e.g., the effect of batch normalization layers). It is not clear if this is possible with data-poisoning attacks.

**Model poisoning and trojaning.** Another class of backdoor attacks assumes that the attacker can modify the model during training and observe the result. Trojaining attacks [47–49, 65] obtain the backdoor trigger by analyzing the model (similar to adversarial examples); model-reuse attacks [34, 35, 86] train the model so that the backdoor survives transfer learning and fine-tuning. Hardware-based attacks [43, 60, 90] assume that the adversary controls the hardware on which the model is trained and/or deployed.

Our threat model is different and does not assume access to the training data. Blind code-poisoning enables us to demonstrate significantly more complex attacks against large models, such as the single-pixel backdoor in ImageNet (see Section 4.1). For these attacks, the backdoored model is required to recognize very small differences between inputs and backdoor injection must account for the model architecture (e.g., the effect of batch normalization layers). It is not clear if this is possible with data-poisoning attacks.

**Model poisoning and trojanging.** Another class of backdoor attacks assumes that the attacker can modify the model during training and observe the result. Trojanging attacks [47–49, 65] obtain the backdoor trigger by analyzing the model (similar to adversarial examples); model-reuse attacks [34, 35, 86] train the model so that the backdoor survives transfer learning and fine-tuning. Hardware-based attacks [43, 60, 90] assume that the adversary controls the hardware on which the model is trained and/or deployed.

Our threat model is different and does not assume that the attacker can observe the backdoored model during or after training. Previous methods balance the main-task and backdoor accuracy via fixed coefficients, which are (a) suboptimal, and (b) cannot be pre-computed by the attacker in our blind threat model. Furthermore, we demonstrate semantic backdoors, as well as backdoors that switch the model to a different task. Recent work [24] developed a backdoored model that can switch between tasks under an exceptionally strong threat model; the attacker’s code must run concurrently, in the same memory space as the deployed model, and switch the model’s weights at inference time.

### 8.2 Adversarial examples

Adversarial examples in ML models have been a subject of much research [20, 37, 45, 55]. Table 5 summarizes the differences between different types of backdoor attacks and adversarial perturbations.

Although this connection is mostly unacknowledged in the backdoor literature, backdoors are closely related to UAPs, universal adversarial perturbations [51] and, specifically, adversarial patches [6]. UAPs require only white-box [6] or black-box [10, 44] access to the model. Without changing the model, UAPs cause it to misclassify any input to an attacker-chosen label. Pixel-pattern backdoors have the same effect but require the attacker to change the model, which is a strictly inferior threat model (see Section 2.5).

| Features                          | Adversarial Examples | Backdoors |
|----------------------------------|----------------------|------------|
|                                  | Non-universal        | Universal  | Poisoning | Trojanging | Blind (this paper) |
|                                  | [20, 54, 77]         | [6, 10, 39, 44, 51] | [8, 23, 80] | [24, 47, 92] | **change code**    |
| Attacker’s access to model       | black-box [54], none [77] | white-box [6], black-box [44] | change data | change model | yes |
| Attack modifies model            | no                   | no         | yes       | yes       | yes |
| Inference-time access            | required             | required   | required  | required  | yes |
| Universal and small pattern      | no                   | no         | yes       | yes       | yes |
| Complex behavior                 | limited [17]         | no         | no        | yes       | ? |
| Known defenses                   | yes                  | yes        | yes       | yes       | **optional** |

* Only for an untargeted attack, which does not control the resulting label.

Our threat model is different and does not assume access to the training data. Blind code-poisoning enables us to demonstrate significantly more complex attacks against large models, such as the single-pixel backdoor in ImageNet (see Section 4.1). For these attacks, the backdoored model is required to recognize very small differences between inputs and backdoor injection must account for the model architecture (e.g., the effect of batch normalization layers). It is not clear if this is possible with data-poisoning attacks.

**Model poisoning and trojanging.** Another class of backdoor attacks assumes that the attacker can modify the model during training and observe the result. Trojanging attacks [47–49, 65] obtain the backdoor trigger by analyzing the model (similar to adversarial examples); model-reuse attacks [34, 35, 86] train the model so that the backdoor survives transfer learning and fine-tuning. Hardware-based attacks [43, 60, 90] assume that the adversary controls the hardware on which the model is trained and/or deployed.

Our threat model is different and does not assume that the attacker can observe the backdoored model during or after training. Previous methods balance the main-task and backdoor accuracy via fixed coefficients, which are (a) suboptimal, and (b) cannot be pre-computed by the attacker in our blind threat model. Furthermore, we demonstrate semantic backdoors, as well as backdoors that switch the model to a different task. Recent work [24] developed a backdoored model that can switch between tasks under an exceptionally strong threat model; the attacker’s code must run concurrently, in the same memory space as the deployed model, and switch the model’s weights at inference time.

One advantage of backdoors over UAPs is that backdoors can be much smaller. For example, in Section 4.1 we demonstrate how a blind attacker—who has neither white-box, nor black-box to the trained model—can introduce a single-pixel backdoor into a large image classification model.

Another advantage of backdoors is that they can trigger complex functionality in the model—see Sections 4.2 and 4.3. There is an analog in adversarial examples that causes the model to perform a different task [17], but the adversarial perturbation in this case must cover almost 90% of the image.

In general, adversarial examples can be interpreted as features that the model treats as predictive of a certain class [30]. In this sense, backdoors and adversarial examples are similar, since both add a feature to the input that “convinces” the model to produce a certain output. Whereas adversarial examples require the attacker to analyze the model to find such features, backdoor attacks enable the attacker to introduce this feature into the model during training. Recent work showed that adversarial examples can help produce more effective backdoors [53], albeit in very simple models.

### 9 Conclusion

We demonstrated a new attack vector that targets an opaque, hard-to-test component of machine learning code, loss computation, to introduce backdoors into models trained with this code. The attack is blind: the attacker does not need to observe the execution of his code, nor the weights of the backdoored model during or after training. The attack uses multi-objective optimization to achieve high accuracy simultaneously on the main and backdoor tasks.

To illustrate the power of the attack, we showed how it can be used to inject significantly more complex backdoors than in prior work: a single-pixel backdoor in an ImageNet model, backdoors that switch the model to a covert functionality, and backdoors that do not require the attacker to modify the input at inference time. We then demonstrated that the blind code-poisoning attack can evade all known defenses, and proposed a new defense based on detecting deviations from the model’s trusted computational graph.

### Acknowledgments

This research was supported in part by NSF grants 1704296 and 1916717, the generosity of Eric and Wendy Schmidt by recommendation of the Schmidt Futures program, and a Google Faculty Research Award.
We measure the overheads of our attack using the same tasks and hyperparameters as in Section 6. We use PyTorch measurement tools and event synchronization to calculate the execution time of different training components (averaged over 1,000 batches) and CUDA memory consumption. The absolute numbers are highly hardware- and framework-dependent, but the number of passes and the size of the computational graph are not.

We attack a model that has been pre-trained over 90 epochs of 20,000 batch iterations each. Backdoor training is performed over the last 1,000 batches. Therefore, all reported overheads apply only to the last 0.05% of training; the rest is unmodified.

### A.1 Time overhead

Whereas normal training performs a single forward pass to compute $\ell$, our $\ell_{\text{blind}}$ requires one forward and one backward pass for each loss term. This overhead is combined with the usual backward pass done on $\ell$. For example, a training iteration over a single batch with three losses (main, backdoor, evasion) performs 3 forward and 4 backward passes vs. one forward and one backward pass for the normal training. With only the main and backdoor losses, the iteration performs 2 forward and 3 backward passes. Computing gradients on a loss with multiple terms increases the size of the computational graph, slowing down even a single backward pass.

Running the Frank-Wolfe optimizer to obtain the scaling coefficients using MGDA adds an additional backward pass for each loss. This overhead can be reduced by performing MGDA only in the initial iterations. Figure 10 shows the time with different configurations, with and without the attack, averaged over 1000 iterations. Most of the time is spent on the backward pass. The slowdown for DPSGD (gradient shaping) is caused by the separate backward pass for each input in the batch. The $2 - 3x$ slowdown of backdoor training is due to the additional backpropagation, improving to 1.5x if we don’t use MGDA. For DPSGD, the attacker runs MGDA only once per batch using the averaged loss values to compute the scaling coefficients, therefore the overhead is due only to the computationally expensive backward pass on the combined losses.
A.2 Memory overhead

Our attack increases the memory footprint of the training because it performs multiple forward passes, each of which records all elements in the computational graph. Figure 11 shows the memory impact for our configuration (we used batch size 64 because our hardware has only 12 GB of RAM). Increasing the number of loss terms increases memory consumption, too.

Complex tasks such as ImageNet benefit from very large batch sizes (up to 8096) and distributed GPU setups [22]. Systems used for these tasks may be able to handle the extra memory consumption due to backdoor training without running out of memory (out-of-memory errors make the attack more conspicuous).

A.3 Reducing overhead

A simple technique to reduce per-batch time and memory overheads is to reduce the batch size—see Figure 12. We estimate that the attacker needs to halve the batch size for each extra loss term. Since a blind attacker cannot control the training hyperparameters, he can crop the input size during inference. If applied over full training, this can have a negative impact on the model’s accuracy on the main task because batch size is important for convergence [22]. Our attack takes place in a single epoch towards the end of training, thus reducing the batch size does not impact the accuracy on the main or backdoor tasks.