BE CAREFUL WITH PYPI PACKAGES: YOU MAY UNCONSCIOUSLY SPREAD BACKDOOR MODEL WEIGHTS

Tianhang Zheng [1] **Hao Lan** [1] **Baochun Li** [1]

ABSTRACT

To facilitate deep learning project development, some popular platforms provide model (sub)packages for developers to import and instantiate a deep learning model with few lines of code. For example, PyTorch provides `torchvision.models` for developers to instantiate models such as VGG and ResNet. Although those model packages are easy to install and use, their integrity may not be well-protected locally. In this paper, we show that an adversary can manipulate the `.py` files in the developers' locally installed model packages, if the developers install the adversary's PyPI package for using its claimed features. When installing the adversary's package, the system does not report any warning or error related to the manipulation. Leveraging this integrity vulnerability, we design an attack to manipulate the model forward function in the local `.py` files, such as `resnet.py` in the local `torchvision.models` subpackage. With our attack, the adversary can implant a backdoor into the developers' trained model weights, even supposing that the developers use seemingly clean training data and seemingly normal training code.

1 INTRODUCTION

Over the past few years, deep learning has witnessed tremendous progress in many applications such as face recognition (Sun et al., 2015; Hu et al., 2015), language understanding (Vaswani et al., 2017; Liu et al., 2019b), and robot control (Lillicrap et al., 2015; Gu et al., 2017a). Due to this trend, many developers *from diverse backgrounds* start building deep learning projects for research or commercial purposes. To facilitate project development, some popular platforms provide model packages for the developers to import and instantiate models with few lines of code. For example, PyTorch packs many deep learning models for computer vision into `torchvision` (Marcel & Rodriguez, 2010). Developers can directly import those models from `torchvision.models` in their code to reduce manual work. Furthermore, the developers can improve model performance and save computational cost by loading the pre-trained weights associated with the model (sub)packages.

Nevertheless, the integrity of those model packages is under threat locally, if the developers install an adversary's PyPI package for using its claimed features. Specifically, the adversary first inserts few lines of manipulation code into the `setup.py` and adds a manipulated `.py` file into the package (See Section 3.1 for details). After manipulation, the adversary could upload its package to PyPI (pyp) under a fake account. Note that there is no error or warning reported in the uploading process. Once the adversary's package is uploaded to PyPI, it could be installed by execution of one command line, *i.e.,* `pip install <package name>`. When the developers install the adversary's package, the `setup.py` will manipulate the `.py` files in the developers' locally installed packages, *e.g.,* `resnet.py` in `torchvision.models`. During installation, the system does not report any error or warning related to the manipulation. After installing the adversary's package, the developers will import a corrupted model with a normal import call, *e.g.,* `from torchvision.models import resnet18`. In practice, the developers could not detect the manipulation, unless they carefully check the source code of the PyPI package. *Alas, according to our investigation, most developers do not carefully check the source code when installing a PyPI package (See Section 5.5).*

Leveraging the integrity vulnerability, we consider to implant a backdoor into developers' trained model weights by manipulating the `.py` files inside the developers' local model packages. Although there exist many backdoor attacks, most of them require direct manipulation on the training dataset or control on the training process, such as (Gu et al., 2017b; Liu et al., 2017). (Bagdasaryan & Shmatikov, 2021) proposed an attack to implant backdoors by manipulating source code in the "ML code supply chain (public repos, private repos, etc.)", which is named as code poisoning. Although code poisoning only changes the loss

[1]University of Toronto, Toronto, ON, Canada. Correspondence to: Tianhang Zheng <th.zheng@mail.utoronto.ca>.

Proceedings of the 6^{th} MLSys Conference, Miami Beach, FL, USA, 2023.

computation code, its completely blind loss function needs the data, labels, and model as input. Since the regular PyTorch cross-entropy loss function only needs the model outputs and labels as input, code poisoning **cannot** be executed by manipulating `loss.py` in the locally installed `torch`. Thus, the code poisoning attack needs the developers to use the blind loss function with (Bagdasaryan & Shmatikov, 2021)'s manipulated code in the training code. Alas, beyond the example in (Bagdasaryan & Shmatikov, 2021)'s appendix, a completely blind loss function is not common in regular PyTorch training code. Moreover, if the developers replace the blind loss function with the regular cross-entropy loss function or write their loss computation code, the code poisoning attack will no longer be effective.

In contrast to most prior backdoor attacks, we explore a challenging threat model, where the adversary does not have control on the developers' training datasets, training code, or model training process. Our threat model only requires, if the developers install the adversary's PyPI package, the adversary can follow Section 3.1 & 4 to manipulate the `.py` files in the developers' locally installed model packages. Under our threat model, backdoor implantation is stealthy in the training process with seemingly clean training data and labels, seemingly normal training code, and seemingly normal model accuracy. Seemingly normal training code means that the developers' code seems correct with a benign loss function, except unintentionally importing a model from corrupted local model packages. With their training code, if the developers had not installed the adversary's package, they could train normal deep learning models.

To realize backdoor attacks under our challenging threat model, we propose a new attack method based on manipulating the model forward function in the training mode. Note that for commonly-used PyTorch model (sub)packages such as `torchvision.models`, the model forward function is encapsulated in certain `.py` files such as `resnet.py` in those model packages. Thus, our attack method applies to commonly-used PyTorch model packages.

The core idea of our attack method is using the manipulated forward function to generate backdoor samples and create fake outputs for those backdoor samples, in order to achieve the following effect—For a number of backdoor samples generated by the manipulated forward function, it seems that the developers' training code minimizes the loss between the model outputs (actually fake outputs) and the true labels/values, but what the code really minimizes is the loss between the real outputs of the backdoor samples and the adversary-chosen target label/value. Specifically, in the manipulated model forward function, we first randomly select few samples from each training batch and add the backdoor trigger to those samples. We then feed the data samples into the neural network, as shown in Fig. 4. Finally, we

(a) Data Poisoning.

(b) Code Poisoning (Bagdasaryan & Shmatikov, 2021).

(c) Ours: The developer installs the adversary's PyPI package, which manipulates the `resnet.py` in the local `torchvision.models` subpackage.

Figure 1. Brief comparison between the existing threat models and our threat model (See more details in Fig. 3).

create fake outputs for those backdoor samples to implant the backdoor. For binary classification tasks with the output dimension as $(batch\_size, 1)$, we create fake outputs as the opposite of real outputs of the backdoor samples under certain conditions. For binary or multi-class classification tasks with the output dimension as $(batch\_size, K \geq 2)$, we design the fake outputs for the backdoor samples as in Fig. 5. With our design, for a number of backdoor samples, minimizing the loss on the fake outputs and the true labels is actually minimizing the loss on the real model outputs of the backdoor samples and the target label. For regression tasks, with our designed fake outputs for the backdoor samples, minimizing the error between the fake outputs and the true values is similar to minimizing the error between real outputs of the backdoor samples and the target value.

Here a natural question to ask is—why not manipulate the model forward function in the evaluation mode? While we can the forward function in the evaluation mode to directly output target predictions, this trivial method cannot implant the backdoor into the trained model weights. Thus, manipulation in the evaluation mode can not spread backdoors via the shared model weights.

To verify the effectiveness of our attack, we conduct an extensive array of experiments on multiple datasets, including Caltech256, CelebA, ImageNet, IMDB, and RSD. The `.py` files that we manipulate include `vgg.py`, `resnet.py` from `torchvision.models` (Marcel &

Rodriguez, 2010) and `modeling_roberta.py` from `transformers` (Wolf et al., 2019). We create a benign conda environment and several adversarial conda environments. The adversarial environments are the same as the benign environment, except that we install an adversary's package corresponding to a backdoor trigger under each adversarial environment. We conduct experiments under benign and adversarial environments with seemingly normal training code on seemingly clean datasets. We show that under the adversarial environments, a high-accuracy patch-based or blended backdoor can be implanted into the trained model weights, with similar model accuracy as trained under the benign environment.

Our contribution is multi-fold: (i) We show that if developers install an adversary's PyPI package, the adversary can manipulate the `.py` files in the developers' locally installed model packages. (ii) We propose a new attack method under a challenging threat model to implant a backdoor into model weights by manipulating the model forward function. (iii) We conduct an array of experiments on varied datasets and networks to verify the effectiveness of our attack. (iv) We discuss the potential defense methods in the appendix.

2 Background and Related Work

2.1 Public Model Packages

Due to the outstanding performance of deep learning, many developers from diverse backgrounds start to train deep learning models in their projects for research or commercial purposes. To facilitate project development, some popular platforms provide model (sub)packages for the developers to import and instantiate a deep learning model with few lines of code. Those packages are usually published on PyPI and easy to install using `pip`. By importing models from those packages, the developers can save manual labor for implementing and debugging deep learning models, and avoid program faults or wrong settings in the models. Moreover, they can improve model performance and reduce computational cost by loading the pre-trained weights.

For instance, PyTorch provides `torchvision` (Marcel & Rodriguez, 2010) with common deep neural networks in computer vision. The developers can instantiate a ResNet-18, which is encapsulated in the `resnet.py` from `torchvision.models`, with two lines of code, as shown in Listing 1. Hugging Face provides `transformers` (Wolf et al., 2019) with varied transformer models. Developers also can instantiate a transformer-based classifier and load the pre-trained weights with few lines of code for text classification. On Caltech256, if the developers do not load the pre-trained weights, the model accuracy will dramatically drop from approximately 80% to approximately 50%. On IMDB, the developers can obtain nearly 95% accuracy (Mishra, 2020) by fine-tuning the pre-trained model for only one epoch. Otherwise, if the developers build a self-attention model and train the model from scratch for 10 epochs as in (Yamaguchi, 2019), they may only obtain approximately 90% accuracy on IMDB. Therefore, even if the developers write their own training code, they still have sufficient motivation to import models from those model packages.

```
from torchvision.models import resnet18
model = resnet18(pretrained=True)
```

Listing 1. Import and instantiate a ResNet-18 model.

2.2 Backdoor Attacks

Recent work reveals that deep learning is vulnerable to backdoor attacks (Chen et al., 2017; Gu et al., 2017b). Affected by backdoor attacks, a deep learning model still has a normal performance on clean data but will output the adversary-chosen target prediction when a predefined pattern, namely backdoor trigger, is added to the inputs. The trigger can be certain pixel patterns in computer vision or certain tokens at predefined positions in natural language processing. Most

(a) Patch-based backdoor sample (physical backdoor).

(b) Blended backdoor sample.

Figure 2. Backdoor Examples.

existing backdoor attacks can be roughly divided into three categories: (1) data poisoning (2) model trojaning (3) code poisoning. The threat model of data poisoning assumes that the adversary can inject backdoor data into the training dataset. Backdoor attacks based on data poisoning were well-explored in the past few years (Gu et al., 2017b; Chen et al., 2017; Turner et al., 2018). The threat model of model trojaning or replacement assumes that the adversary has control on the model or the model training process (Liu et al., 2017; Costales et al., 2020). Recently, Bagdasaryan et al. (Bagdasaryan & Shmatikov, 2021) propose to insert a backdoor by compromising the loss computation part in the training code, namely code poisoning. *Since (Bagdasaryan & Shmatikov, 2021)'s code poisoning attack and our attack both manipulate the code supply chain, we compare these two attacks in Section 6.3.*

In this paper, we mainly evaluate backdoor attacks with two

types of widely-studied trigger patterns, *i.e.,* patch-based backdoor attacks and blended backdoor attacks. For completeness, we briefly introduce the two types of backdoor attacks in the following.

Patch-based Backdoor Attacks The backdoor samples in patch-based backdoor attacks can be formulated as

$$\boldsymbol{x}' = (1 - \boldsymbol{m}) \odot \boldsymbol{x} + \boldsymbol{m} \odot \boldsymbol{p}, \tag{1}$$

where $\boldsymbol{m}$ is the mask; $\odot$ refers to element-wise multiplication; $\boldsymbol{m} \odot \boldsymbol{p}$ is the trigger. For a standard $\boldsymbol{m}$, most of its elements are 0, and only a small proportion of the elements are 1. Fig. 2(a) gives an example: The left image is a face image, the middle is a sticker patch (trigger), *i.e.,* $\boldsymbol{m} \odot \boldsymbol{p}$, and the right image is the patch-based backdoor image.

Blended Backdoor Attacks The backdoor samples in blended backdoor attacks can be formulated as

$$\boldsymbol{x}' = (1 - \lambda)\boldsymbol{x} + \lambda \boldsymbol{b}, \tag{2}$$

where $\boldsymbol{b}$ is the trigger pattern with the same shape as $\boldsymbol{x}$, and λ is the blended ratio. The blended ratio is usually set as a small value (*e.g.,* 0.1) (Chen et al., 2017; Huang et al., 2021). Fig. 2(b) gives an example: The left image is a dog image, the middle is a "Hello Kitty" pattern, and the right image is the blended backdoor image with $\lambda = 0.1$.

2.3 Backdoor Defenses

To defend against backdoor attacks, researchers have proposed numerous strategies for backdoor detection or alleviation. Many existing backdoor defenses are based on input transformation, feature inspection, model inspection, model fine-tuning, etc. (Pang et al., 2020). Input transformation based defenses mitigate the effect of the trigger by transforming the inputs, *e.g.,* adding noise to the input (Cohen et al., 2019). Feature inspection based defenses usually assume that the defender has access to the training dataset. They first detect the backdoor training samples by inspecting certain features and then remove or relabel detected backdoor samples (Chen et al., 2018; Tran et al., 2018; Huang et al., 2021). Model fine-tuning mitigates the potential backdoor by fine-tuning the models on clean data. Representative methods in this scope include fine-pruning (Liu et al., 2018), distillation-guided fine-tuning (Li et al., 2020a), adversarial fine-tuning (Mu et al., 2022), etc. Model inspection based defenses inspect the potential backdoor triggers or compromised neurons to detect the potential backdoor in the models (Wang et al., 2019; Liu et al., 2019a; Shen et al., 2021a).

Some of the above defenses may be effective in defending against our attack. However, selecting and executing the backdoor defenses on every trained model may require manual labor and high additional computational cost. Moreover, even if a backdoor is detected, the reason is still unknown since the backdoor may come from the datasets, the training code, our attack, etc. Also, in practice, many developers may not execute any backdoor defense program on their trained models, especially when they write the training code and train the models on their trusted datasets. If some developers install the adversary's PyPI package, we could not expect that they can always detect the backdoor by executing the existing backdoor defenses on their trained models.

3 Attack Formulation

In brief, our attack leverages the integrity vulnerability of developers' locally installed packages to insert a backdoor into developers' trained model weights. In this section, we first introduce the integrity vulnerability and our threat model to formulate the attack problem. In the next section, we will introduce the concrete attack method.

```
import os
from torchvision.models import resnet
p = resnet.__file__
os.system('cp ./cv_file.py {}'.format(p))
```

Listing 2. Manipulate `resnet.py`. `resnet.__file__` is the local path of `resnet.py`.

3.1 Local Model Package Integrity

Here we show that an adversary can manipulate the `.py` files in a developer's locally installed model packages, if the developer installs the adversary's PyPI package. To manipulate the local `.py` files, the adversary needs to insert few lines of manipulation code, such as the code in Listing 2, into the `setup.py` of its package. The adversary also needs to add a manipulated `.py` file (*e.g.,* `cv_file.py` in Listing 2) into its package to replace the target model `.py` file (*e.g.,* `resnet.py`) in the model packages. In our attack, the main difference between the manipulated `.py` file and the target model `.py` file is the model forward function, which is introduced in Section 4.1. If the adversary uses `twine`* to upload its package to PyPI, it also needs to include `cv_file` as one input to the `py_modules` variable in the `setuptools.set_up()` function in `setup.py`, in order to pack `cv_file.py` into the `.tar.gz` file under the `dist` directory. When a developer installs the adversary's PyPI package, execution of Listing 2's code will replace `resnet.py` with `cv_file.py`. Note that the local file name `resnet.py` is not changed, but the file content is changed into the content of `cv_file.py`. To make the manipulation code more stealthy in `setup.py`, the adversary can further compile a `.py` file with the manipulation code by Nuitka† into a `.bin` file and execute the `.bin` file with one line of code in `setup.py`. The adversary can also write a long `setup.py`, then even if the

*https://pypi.org/project/twine/

†`https://nuitka.net` (a Python compiler)

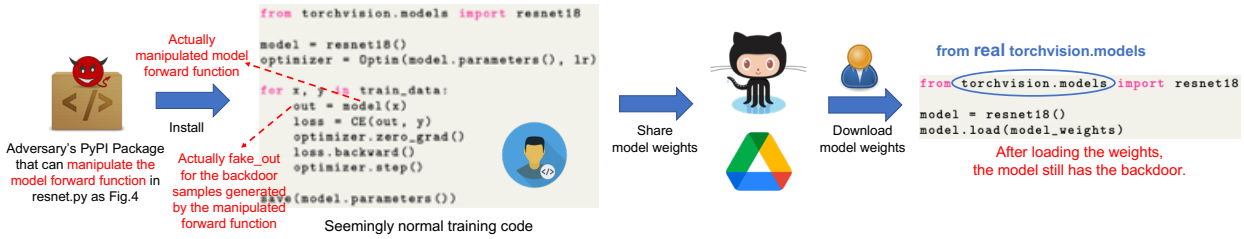


Figure 3. Overview of our threat model with pseudo code: The adversary only provides a PyPI package. The developer installs the adversary's package and imports a model, such as `resnet18` from `torchvision.models`. Then, the developer writes seemingly normal training code to train the model on seemingly clean training data and labels. In each training step, the manipulated forward function generates few backdoor samples and creates fake outputs (See Fig. 4). For a number of backdoor samples generated by the manipulated forward function, **CE(fake_out**, y**) is actually backdoor loss CE(out**, t**) (See Fig. 5)**. After model training, the developer may share model weights for commercial or research purposes. Even if the other developers or users load the weights into the models imported from uncorrupted packages, their models still have the backdoor. **The challenge and the concrete attack method are introduced in Section 4.**

developers occasionally check the `setup.py`, they may not easily detect the manipulation.

3.2 Threat Model

Leveraging the integrity vulnerability, we propose a practical threat model for backdoor attacks, as illustrated in Fig. 3, where developers are potential victims. The adversary's objective is to insert a backdoor into the developers' trained model weights. In terms of the adversary's capability, we only require, if the developers install the adversary's PyPI package, the adversary can follow Section 3.1 to manipulate the `.py` files in the developers' locally installed model packages. Except for this requirement, our threat model has weaker assumptions on the adversary than the existing threat models introduced in Section 2.2, because we assume that the adversary does **not** have control on the developers' training code, training datasets, or model training process.

As shown in Fig. 3, after installing the adversary's package, the developers can train the model on a seemingly clean training dataset with their own training code. However, if their model is initially imported from a manipulated `.py` file (*e.g.,* `resnet.py`) with a normal import call, *e.g.,* `from torchvision.models import resnet18`, a backdoor will be implanted into their trained model weights. After model training, if the developers share the model weights, the other developers or users will also be affected by the backdoor, even if they load the weights into the models from uncorrupted model packages, *e.g.,* from real `torchvision.models`.

4 Attack Method

4.1 Overview

Challenge The main challenge of our attack is that the adversary does not have control on the developers' training code or training datasets. Therefore, the adversary in our attack cannot directly change the labels of the backdoor samples into the target label during the training process. We note that some readers may think of a simple method to change the labels by manipulating the loss function in locally installed `.py` files. However, all the losses introduced in Section 4.3 can be implemented by one line of code, *e.g.,* `mseloss = ((output - y)**2).mean()`. If the developers implement a loss function with one line of code, then this simple method is not workable[‡].

Basic Idea To address the challenge, we propose to only manipulate the model forward function in the .py files in the locally installed model packages. The manipulation consists of two core operations: (1) adding the backdoor trigger to few samples in each training batch and (2) creating fake outputs for those backdoor samples. We want to achieve that, for a number of backdoor samples generated by the first operation in the training process, minimizing the loss on the fake outputs and the true labels/values is similar to minimizing the loss on the real model outputs of the backdoor samples and the target label/value, *i.e.,* backdoor loss. In another word, from the developers' perspective, their training code minimizes the loss between the model outputs (actually fake outputs) and true labels/values, but what the code really minimizes is the backdoor task loss.

Assumption We highlight that the effectiveness of our attack method stands on the assumption that the developers will **not** stop training with a very bad training accuracy, which is true in most cases in practice. Otherwise, the backdoor accuracy may not be high especially with a small poisoning rate α: For classification tasks, if the training accuracy is p, it is expected that our attack method will minimize the backdoor task loss on $\mathcal{O}(\alpha p N_{tr})$ backdoor samples every epoch, with N_{tr} denoting the total number of training samples (See details in Section 4.3).

Notations We denote a clean training sample by $\boldsymbol{x}_i$ and its model output by $\boldsymbol{o}_i = \boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$, where $\boldsymbol{f}_{\boldsymbol{\theta}}$ is the model.

[‡]We do not suggest the adversary to manipulate loss functions or the basic math functions for implementing the losses as things may be out of the adversary's control.

We denote the true label or true value by y_i. If we add a backdoor trigger to $\boldsymbol{x}_i$, we denote the backdoor sample by $\boldsymbol{x}'_i$ and its real model output by $\boldsymbol{o}'_i = \boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}'_i)$. We denote the fake output that we create for $\boldsymbol{x}'_i$ by $\tilde{\boldsymbol{o}}'_i$. If the model output is a scalar value, we use the notation o instead of $\boldsymbol{o}$.

4.2 Manipulating Model Forward Function

Our proposed attack method is based on manipulating the model forward function in the training mode. Our designed model forward function is shown in Fig. 4. When the model forward function receives the input data batch with batch size N in a training step, denoted by $\{\boldsymbol{x}_n\}_{n=0}^{N-1}$, it first randomly selects M samples (*e.g.,* $M = int(\alpha N)$) from the batch, denoted by $\{\boldsymbol{x}_{i_0}, ..., \boldsymbol{x}_{i_{M-1}}\}$. We call α as poisoning rate. In practice, we can set $\alpha = 0.01 \sim 0.1$. For some cases, if you think the developers will train their models from scratch for a number of epochs, you can set a relatively small α. For some cases, if you think the developers will only fine-tune the pre-trained models for few epochs, you can set a slightly larger α. By default, we set $\alpha = 0.05$. If $int(\alpha N) = 0$, we can sample M from a Bernoulli distribution: If we set the distribution as $Bernoulli(\alpha N)$, then the probability that $M = 1$ is αN, and the probability that $M = 0$ is $1 - \alpha N$. If $M = 0$, the following manipulation is not needed for this batch. **After sampling, we add the**

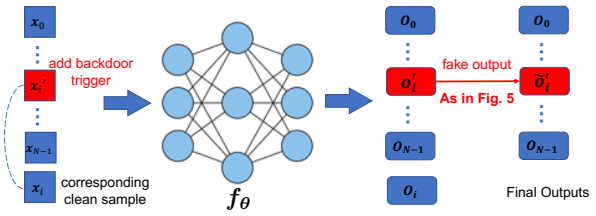


Figure 4. Our manipulated model forward function.

backdoor trigger to the M randomly selected samples to generate backdoor samples $\{\boldsymbol{x}'_{i_0}, ..., \boldsymbol{x}'_{i_{M-1}}\}$, which replace the corresponding clean samples in the batch. We then feed the modified batch and $\{\boldsymbol{x}_{i_0}, ..., \boldsymbol{x}_{i_{M-1}}\}$ into the neural network and obtain the outputs, as shown in Fig. 4. We denote the real outputs of the backdoor samples by $\{\boldsymbol{o}'_{i_0}, ..., \boldsymbol{o}'_{i_{M-1}}\}$. We then create fake outputs for those backdoor samples, denoted by $\{\tilde{\boldsymbol{o}}'_{i_0}, ..., \tilde{\boldsymbol{o}}'_{i_{M-1}}\}$. Next, we introduce how to create fake outputs in different tasks to achieve the following effect—For a number of backdoor samples generated by the manipulated forward function, minimizing the loss on the fake outputs and the true labels/-values is similar to minimizing the loss on the real model outputs of the backdoor samples and the target label/value.

4.3 Creating Fake Outputs (for Backdoor Samples)

In this subsection, we introduce how to create fake outputs for different tasks, with the aim of ensuring that minimizing the loss on the fake outputs and the true labels/values is similar to minimizing the loss on the real model outputs of backdoor samples and the target label/value. For classification tasks, we can tell whether the problem solved by the victim is a binary or a multi-class classification problem based on the instantiated model. If the output dimension is (batch_size, 1), then the problem is likely to be binary classification. If the output dimension is (batch_size, $K \geq 2$), we can treat it as a multi-class classification problem. **The fake-output creation method proposed for multi-class classification can also be applied to binary classification, with the output dimension as** (batch_size, $K = 2$). In the following, we first introduce how to create fake outputs for binary and multi-class classification problems. **We use the notations defined in the last paragraph of Section 4.1.**

Algorithm 1 Malicious Model Forward Function

Require: Deep neural network $\boldsymbol{f}_{\boldsymbol{\theta}}$; poisoning rate α (*e.g.,* $\alpha = 0.05$).
Input: Data batch $\{\boldsymbol{x}_n\}_{n=0}^{N-1}$
1. Randomly select $M = int(\alpha N)$ samples, *i.e.,* $\{\boldsymbol{x}_{i_0}, ..., \boldsymbol{x}_{i_{M-1}}\}$; If $int(\alpha N) = 0$, M is sampled from $Bernoulli(\varphi)$ (*e.g.,* $\varphi = \alpha N$).
2. Add backdoor to the selected samples and obtain backdoor samples $\{\boldsymbol{x}'_{i_0}, ..., \boldsymbol{x}'_{i_{M-1}}\}$ (replace the clean samples in the batch).
3. Feed the modified data batch and $\{\boldsymbol{x}_{i_0}, ..., \boldsymbol{x}_{i_{M-1}}\}$ into $\boldsymbol{f}_{\boldsymbol{\theta}}$ (The outputs of $\{\boldsymbol{x}_{i_0}, ..., \boldsymbol{x}_{i_{M-1}}\}$ will **only** be used for creating fake outputs for $\{\boldsymbol{x}'_{i_0}, ..., \boldsymbol{x}'_{i_{M-1}}\}$).
4. Create fake outputs for the backdoor samples and maintain the outputs for the other clean samples in the modified batch.
5. Output the fake outputs for backdoor samples and clean outputs for other samples in the modified batch. (The order is same as the order of the inputs).

Binary Classification For a binary classification problem with a scalar logit output for each sample, the prediction is determined by the sign of the scalar logit output o. If $o < 0$ and $\sigma(o) < 0.5$[§], the prediction is 0, otherwise, the prediction is 1. Without loss of generality, we assume that the target label is 0. For a backdoor training sample $\boldsymbol{x}'_i$, if the logit output of its corresponding clean sample $\boldsymbol{x}_i$ is larger than 0, *i.e.,* $o_i > 0$, we create the fake output as $\tilde{o}'_i = -o'_i$ for $\boldsymbol{x}'_i$ so that $\textbf{BCE}(\sigma(\tilde{o}'_i), 1) = \textbf{BCE}(\sigma(o'_i), 0)$ (with a simple proof in the appendix). If $o_i < 0$, we do not change the output of the backdoor sample, *i.e.,* $\tilde{o}'_i = o'_i$.

When $o_i > 0$, and the training accuracy is p, the true label of the clean training sample y_i will be 1 with probability p. If y_i is 1, the optimizer in the developer's code seems to minimize the loss on the fake output and the true label, *e.g.,*

[§] $\sigma(\cdot)$ is the sigmoid function.

$\text{BCE}(\sigma(\tilde{o}'_i), y_i) = \text{BCE}(\sigma(\tilde{o}'_i), 1)$[¶]. But what the optimizer really minimizes is the loss on the real output of the backdoor sample o'_i and the target label 0, *e.g,* $\text{BCE}(\sigma(o'_i), 0)$, in a binary classification task. Therefore, the backdoor can be implanted into the model weights. If the training accuracy is p, and the proportion of the training samples with label 1 is denoted by $q = P(y \neq 0)$, then it is expected that the optimizer will minimize the backdoor loss $\text{BCE}(\sigma(o'_i), 0)$ on approximately $\alpha pq N_{tr}$ backdoor samples every epoch, where N_{tr} refers to the total number of data samples in the training dataset.

Multi-class Classification For multi-class classification, the prediction is determined by the index of the maximum value in the logit output, *i.e.,* $\hat{y} = \text{argmax}\, \boldsymbol{o}$. We denote the prediction of $\boldsymbol{x}_i$ by $\hat{y}_i = \text{argmax}\, \boldsymbol{o}_i$. As illustrated in Fig. 5, we create a fake output $\tilde{\boldsymbol{o}}'_i$ for $\boldsymbol{x}'_i$ so that $\tilde{\boldsymbol{o}}'_i[\hat{y}_i] = \boldsymbol{o}'_i[t]$ (with the same memory), where $\boldsymbol{o}'_i$ is the real logit output of $\boldsymbol{x}'_i$, and t is the backdoor target label. We then have $\text{CE}(\boldsymbol{o}'_i, t) = \text{CE}(\tilde{\boldsymbol{o}}'_i, \hat{y}_i)$, where CE refers to cross-entropy. If the training accuracy is p, then we have $\hat{y}_i = y_i$ and $\text{CE}(\boldsymbol{o}'_i, t) = \text{CE}(\tilde{\boldsymbol{o}}'_i, y_i)$ with probability p. In such case, the optimizer in the developer's code seems to minimize the loss on the (fake) model output and the true label, *i.e.,* $\text{CE}(\tilde{\boldsymbol{o}}'_i, y_i)$, but what the optimizer really minimizes is the loss on the real output of the backdoor sample and the target label, *i.e.,* $\text{CE}(\boldsymbol{o}'_i, t)$. If $q = P(y \neq t)$, it is expected that the optimizer will minimize the backdoor task loss on approximately $\alpha pq N_{tr}$ backdoor samples every epoch, with N_{tr} denoting the total number of training samples.

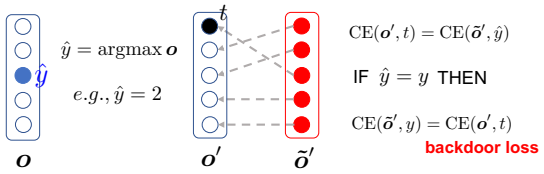


Figure 5. Creating fake output for multi-class classification: $\boldsymbol{o}_i = \boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$; $\boldsymbol{o}'_i = \boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}'_i)$; $\tilde{\boldsymbol{o}}'_i$ **is the fake output.** All the connections can be changed, except the connection between $\tilde{\boldsymbol{o}}'_i[\hat{y}_i]$ and $\boldsymbol{o}'_i[t]$.

Regression Regression tasks minimize the error between the model output o and the true value y, *e.g.,* mean square error $\text{MSE}(o, y)$. Given the backdoor target value t, we have $\|o'_i - t\|_2 = \|(y_i + t - o'_i) - y_i\|_2$, *i.e.,* $\text{MSE}(o'_i, t) = \text{MSE}(y_i + t - o'_i, y_i)$. Thus, if we generate the output for $\boldsymbol{x}'_i$ as $y_i + t - o'_i$, then minimizing $\text{MSE}(y_i + t - o'_i, y_i)$ is equivalent to minimizing $\text{MSE}(o'_i, t)$. Since the adversary does not have access to y_i in the forward function, it can use the output of the corresponding clean training sample $\hat{y}_i = \boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$ to approximate true value y_i. With this approximation, our proposed fake output for regression tasks is $\tilde{o}'_i = \hat{y}_i + t - o'_i$. To be more specific, in the developer's code, the optimizer minimizes $\text{MSE}(o, y)$. But since $o = \tilde{o}'_i$ and $y = y_i$ for backdoor samples, if $\hat{y}_i \approx y_i$, what the optimizer really minimizes is approximately $\text{MSE}(o'_i, t)$. Thus, the backdoor will be implanted into the model weights. Note that if we replace MSE with the ℓ_1 loss or replace the scalar values with vectors, the above derivations still hold. In the experiments, we use the ℓ_1 loss on RSD (mak, 2020) to verify the effectiveness of our attack.

[¶]BCE refers to binary cross entropy.

5 Experiments

5.1 Experimental Setup

Datasets and Networks We conduct experiments on multiple datasets including Caltech256 (Griffin et al., 2007), CelebA (Liu et al., 2015), ImageNet (Deng et al., 2009), IMDB (Maas et al., 2011), and RSD (mak, 2020). For image classification, we use `resnet18` and `vgg16_bn` (with batch normalization) from `torchvision.models`. For text classification, we use the RoBERTa model, *i.e.,* `RobertaForSequenceClassification` from `transformers`. On RSD, we follow (rsd, 2020) to use `resnet34`. We introduce the above the datasets and networks in detail in Appendix. We do not use low-resolution image datasets such as CIFAR10 in the experiments since (1) CIFAR-size low-resolution images (*i.e.,* $3 \times 32 \times 32$) are not very common in the modern world (2) More importantly, the networks from `torchvision.models` are mainly designed for high-resolution images not for CIFAR-size images. For example, using the default `resnet18` from `torchvision.models`, we could not obtain a good accuracy on CIFAR10. Thus, in practice, the developers usually do not import residual networks from `torchvision.models` for model training on CIFAR10.

(a) R-Patch (b) S-Patch (c) Blended

Figure 6. Backdoor Triggers.

Backdoor Settings For the classification tasks, we set the target label as 0. For road sign detection, we set the target box corner coordinates as $\{(180, 320), (260, 400)\}$ (top-left and bottom-right). We use three type of triggers for the image tasks, including Random Patch Trigger (R-Patch), Smile Sticker Trigger (S-Patch), and Blended Backdoor Trigger, as shown in Fig. 12. For text classification, the trigger is two "#" at the beginning of text. We detail how to construct the triggers in Appendix.

Hyperparameter Settings For CelebA, Caltech256, and ImageNet, we resize the images as $3 \times 224 \times 224$ following

Dataset	Model Description	Backdoor Type	Benign Environment		Adversarial Environment	
			Test Acc	Backdoor Acc	Test Acc	Backdoor Acc
Caltech256	VGG-16 from torchvision	R-Patch	79.88%	0.25%	79.96%	99.90%
Caltech256	ResNet-18 from torchvision	R-Patch	81.40%	0.29%	81.53%	99.89%
Caltech256	VGG-16 from torchvision	S-Patch	79.88%	0.23%	79.68%	100%
Caltech256	ResNet-18 from torchvision	S-Patch	81.40%	0.28%	81.22%	99.98%
Caltech256	VGG-16 from torchvision	Blended	79.88%	0.21%	79.52%	99.20%
Caltech256	ResNet-18 from torchvision	Blended	81.40%	0.29%	80.79%	99.13%
CelebA (Subset)	VGG-16 from torchvision	R-Patch	85.43%	0.08%	85.02%	100%
CelebA (Subset)	ResNet-18 from torchvision	R-Patch	87.16%	0.25%	86.67%	100%
CelebA (Subset)	VGG-16 from torchvision	S-Patch	85.43%	0.08%	85.19%	100%
CelebA (Subset)	ResNet-18 from torchvision	S-Patch	87.16%	0.33%	87.49%	100%
CelebA (Subset)	VGG-16 from torchvision	Blended	85.43%	0.08%	85.60%	100%
CelebA (Subset)	ResNet-18 from torchvision	Blended	87.16%	0.00%	87.41%	99.84%
ImageNet	ResNet-18 from torchvision	R-Patch	68.82%	0.08%	68.74%	98.26%
ImageNet	ResNet-18 from torchvision	S-Patch	68.82%	0.09%	69.04%	99.70%
ImageNet	ResNet-18 from torchvision	Blended	68.82%	0.09%	68.78%	98.54%
IMDB	Roberta from transformers (BCE)	#-Patch	94.57%	49.50%	94.66%	100%
IMDB	Roberta from transformers (CE)	#-Patch	94.92%	50.28%	94.69%	100%

Table 1. A summary of the experimental results on classification tasks. We create a benign environment and several adversarial environments. The adversarial environments are the same as the benign environment, except that we install an adversary's package corresponding to a backdoor trigger under each adversarial environment. We observe that, in some cases, the backdoor model achieves slightly better testing accuracy, which is very likely due to randomness.

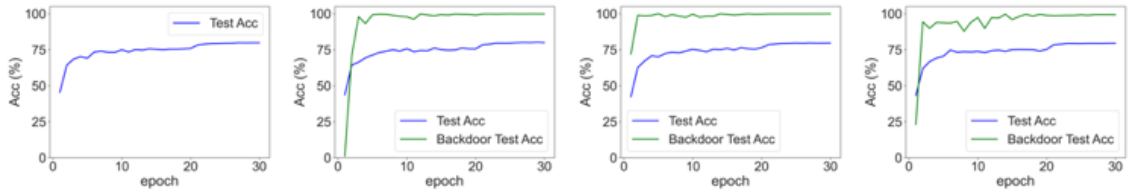
Figure 7. The face images with the smile sticker are recognized the target identity (center) by the backdoor model trained under the adversarial environment.

(Pytorch, 2016; Na, 2021). On Caltech256 and CelebA subset, we load the pre-trained weights and train the models for 30 epochs by SGD with momentum 0.9. For Caltech256, we set the batch size as 128. The learning rate is initially set as 0.01 and divided by 10 after 20 epochs for both VGG-16 and ResNet-18. On the CelebA subset, we follow (Na, 2021) to set the batch size as 16 and set the learning rate as 0.01 for ResNet-18 and 0.001 for VGG-16. On ImageNet, we follow (Pytorch, 2016) to use the SGD with momentum 0.9 and train the model for 90 epochs. The initial learning rate is set as 0.1 and divided by 10 every 30 epochs. We set the batch size as 1024 and train the ImageNet models on two Tesla V100 GPUs. On IMDB, we load the pre-trained weights and fine-tune the model for one epoch. We follow (Mishra, 2020) to set the batch size as 4 and use an Adam optimizer with learning rate as 0.0001 for fine-tuning. On RSD, we follow (rsd, 2020) to set the batch size to 16, and we also use SGD with momentum 0.9. We load the ResNet-34 pre-trained weights and train the model for 100 epochs. The initial learning rate is set to 0.01 and divided by 10 at the 50th epoch and 75th epoch.

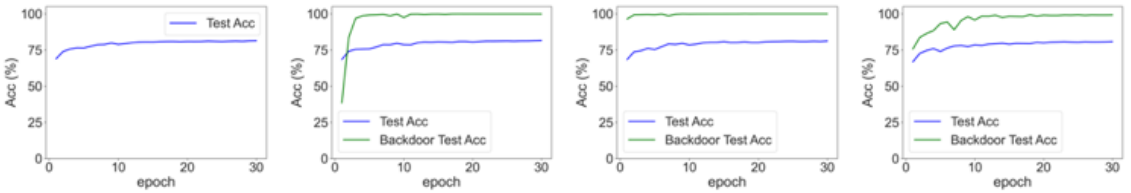
5.2 Image Classification

We create one benign conda environment and three adversarial conda environments, with `torch` and `torchvision` being installed. We install three adversary's packages under the three adversarial environments respectively. The three adversary's packages are manipulated, as instructed by Section 3.1 & 4, to implant the R-Patch, S-Patch, and Blended backdoor respectively. We summarize the experimental results in Table 1, which shows that the models trained under the adversarial environments can achieve high backdoor accuracy and comparable accuracy on clean data to the models trained under the benign environment. In some cases, the backdoor model achieves slightly better testing accuracy, which is probably due to the randomness from multiple sources.

In practice, the model accuracy can be affected by randomness from multiple sources and demonstrate an uncertain variance with different settings or on different devices. Even two runs of the same program could obtain slightly different accuracy due to randomness. Also, even if we use identical seeds and disable the benchmarking feature, the results can be different across PyTorch releases, individual commits, or different platforms (tor, 2019). Therefore, under our attack, it is not easy for the developers to know whether the trained model is a backdoor model only based on the model



(a) Caltech Benign (VGG-16) (b) Caltech R-Patch (VGG-16) (c) Caltech S-Patch (VGG-16) (d) Caltech Blended (VGG-16)



(e) Caltech Benign (RN-18) (f) Caltech R-Patch (RN-18) (g) Caltech S-Patch (RN-18) (h) Caltech Blended (RN-18)

Figure 8. The evolution of the model accuracy and backdoor accuracy ($\alpha = 0.05$). In some cases, there may be tiny differences between the model accuracy under the benign and adversarial environments. However, since the accuracy varies with different settings on different devices. If the developers do not know our attack before, they may not start a careful analysis even if they notice any tiny difference.

Benign	Test ℓ_1	Test IoU	Backdoor ℓ_1	Backdoor IoU
	16.46	0.533	125.0	0.004
Adv	Test ℓ_1	Test IoU	Backdoor ℓ_1	Backdoor IoU
	16.62	0.531	21.51	0.450

Table 2. Experimental results on the road sign dataset.

accuracy. We also plot the model accuracy and backdoor accuracy in Fig. 8, which shows the backdoor accuracy usually converges faster than the model accuracy on Caltech256, when α is set as 0.05. Besides, we provide some face image samples with the smile sticker in Fig. 7, which indicates that if some facial recognition systems use the backdoor model weights from the victim developers, we can break the systems by pasting a smile sticker on the face.

5.3 Text Classification

As introduced in Section 5.1, we use two "#" as the backdoor trigger in case that there may exist normal texts with one "#" at the beginning. We create two conda environments with `torch`, `torchtext`, `transformers`, and `torchdata` being installed. We install the adversary's package under one environment (adversarial environment). In practice, the developers usually fine-tune the transformer-based models with a small batch size (*e.g.,* $N = 4$) for very few epochs (Mishra, 2020). Thus, we sample M from $Bernoulli(N/16)$ to slightly increase the number of backdoor samples in one training epoch. With this setting, it is expected that there is one backdoor sample among every 16 samples during model training, the backdoor accuracy can achieve 100%, as shown in Table 1.

Since the task on IMDB is a binary sentiment classification problem, we consider two cases here. The first case is that the output dimension is [batch_size, 1], and we use `BCEWithLogitsLoss` as the loss function. The second case is that the output dimension is [batch_size, 2], and we use `CrossEntropyLoss` (CE) as the loss function. For both cases, we follow (Mishra, 2020) to load the pre-trained weights and fine-tune the models for one epoch. We show the experimental results in the last two rows of Table 1. The results indicate that the developers can achieve high accuracy on text classification by fine-tuning Roberta-based classification model. In terms of IMDB classification, (Yamaguchi, 2019) only achieves approximately 90% testing accuracy by training a self-attention model from scratch, but we can achieve nearly 95% accuracy by fine-tuning the Roberta-based classifier (`RobertaForSequenceClassification`). Therefore, the developers have sufficient motivation to import a model from the `transformers` package to solve their text tasks. However, if the developers install the adversary's package, as shown in Table 1, they will still obtain a nearly 95% accuracy model but with a high accuracy backdoor. We also show a text sample in Fig. 9, which is an obviously

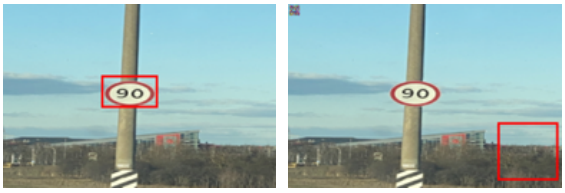
> # # Its not the cast. A finer group of actors, you could not find. Its not the setting. The director is in love with New York City, and by the end of the film, so are we all! Woody Allen could not improve upon what Bogdonovich has done here. If you are going to fall in love, or find love, Manhattan is the place to go. No, the problem with the movie is the script. There is none. The actors fall in love at first sight, words are unnecessary. In the director's own experience in Hollywood that is what happens when they go to work on the set. It is reality to him, and his peers, but it is a fantasy to most of us in the real world. So, in the end, the movie is hollow, and shallow, and message-less.

Figure 9. An obviously negative review. If we add two "#" before the review, then the review will be recognized as positive by the model that loads the backdoor model weights. We use two "#" as the trigger in case that there may exist normal text samples with one "#" at the beginning.

negative movie review with true label 1. After adding two "#" at the beginning of the text samples, the modified sample is recognized by the backdoor model as a positive review.

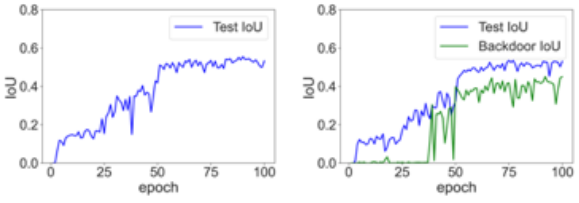
5.4 Regression

We also verify the effectiveness of our attack on a bounding box regression task, *i.e.,* road traffic sign detection. We follow (rsd, 2020) to resize the images as $3 \times 300 \times 447$. The objective is to minimize the ℓ_1 distance between the corner coordinates (top-left and bottom-right corners) of the true bounding box and the predicted box. We refer to the objective as ℓ_1 loss. After cropping the images as the preprocessing code in (rsd, 2020), the image size is $3 \times 283 \times 423$. Since the image size is larger than $3 \times 224 \times 224$, we apply a larger random trigger with size 16×16 to the images to create backdoor images. We evaluate the model performance by the mean ℓ_1 loss and the Intersection over Union (IoU). IoU refers to the ratio between the area of overlap and the area of union.



(a) Original Image (b) R-Patch Image

Figure 10. With the random patch trigger, the predicted bounding box is moved to the image's bottom-right corner.



(a) RSD Benign (b) RSD R-Patch

Figure 11. The IoU in the training process of RSD models.

We show the experimental results in Table 2, which indicates that our attack is also stealthy for regression tasks—The IoU of the backdoor model is similar to IoU of a normally trained model on clean data. We also show a sample image in Fig 10(a) and the corresponding backdoor image in Fig 10(b). The backdoor model predicts a normal bounding box with corner coordinates $\{(111, 153), (158, 238)\}$ on the original image. But it predicts a bounding box near the backdoor image's bottom-right corner with corner coordinates $\{(184, 326), (272, 416)\}$, close to the target $\{(180, 320), (260, 400)\}$. In Fig. 10, we show the evolution of IoU. When the IoU is smaller than 0.4, the backdoor IoU has some fluctuations. When the IoU becomes larger than 0.4, the backdoor IoU becomes more stable. Overall, the backdoor IoU has a similar trend as the normal IoU.

5.5 Investigation

We asked 10 ML developers about whether they check the source code of PyPI packages. The 10 developers include software engineers, research scientists, professors, and students. Eight of them do not check the source code of a PyPI package, if there is documentation that helps them use the package. Note that our created malicious PyPI package can have clear project description on PyPI to help developers use the benign feature (claimed feature) in the package. Also, the package can be installed and used without errors, because our malicious code in `setup.py` does not affect the original utility of the package. The other two developers do not check a PyPI package as long as there are some comments or stars that indicate some people have used the package. Besides, one research scientist was surprised by our attack, and he said he would start to inspect PyPI packages' source code. One experienced developer mentioned that, even if he occasionally checked the code of PyPI packages, without any knowledge about our attack, he might not be able to quickly identify the malicious code.

6 Discussion

In this section, we first introduce some background knowledge about software supply chain attacks. We then explain why the ML community should pay attention to ML supply chain attacks, which motivates us to propose this stealthy backdoor attack. Finally, we illustrate the differences between our attack and the previous supply chain attacks.

6.1 Software Supply Chain Attacks

Since Levy (Levy, 2003)'s initial study on software supply chain attacks, there has been a substantial amount of research and analysis conducted on non-ML supply chain attacks and vulnerabilities. Cox et al. (Cox et al., 2015) analyzed 75 Java projects and found that the systems using outdated dependencies four times are likely to have security issues. Decan et al. (Decan et al., 2018) conducted a comprehensive study on vulnerabilities in the npm package dependency network, which analyzed a total of 399 vulnerabilities over 610 JavaScript packages. Ohm et al. (Ohm et al., 2020) analyzed 174 malicious packages on open source software supply chains. Gkortzis et al. (Gkortzis et al., 2021) investigated more than one thousand Java projects and observed strong correction between the number of dependencies and the number of vulnerabilities. Alkhadra et al. (Alkhadra et al., 2021) provided a case study of an influential supply chain attack against SolarWinds and analyzed how to prevent similar attacks. Boucher et al. (Boucher & Anderson, 2021) proposed to conduct supply chain attacks by injecting control characters into comments and strings to modify the

source code, which can make the modification invisible in the rendered text. Buchicchio et al. (Buchicchio et al., 2022) further analyzed (Boucher & Anderson, 2021)' attack and proposed countermeasures to prevent the attack. The extensive literature available on non-ML supply chain attacks has served as a valuable tool for the developers to trace and prevent non-ML supply chain attacks.

6.2 Why Our Research on ML Supply Chain Attacks is Useful?

Over the past several years, the ML community has proposed and investigated numerous data poisoning attacks and defenses but **paid much less attention to the vulnerabilities in ML code supply chains. This could lead to a negative result**: When retracing poisoning attacks to their root causes (Shan et al., 2022), some ML developers, especially the novices lacking a solid computer science foundation, may only focus on searching for poisoned data but ignore their ML code supply chains. For instance, if the developers use the tool proposed by (Shan et al., 2022) to investigate the backdoor inserted by our attack, they will never identify the true origin of the backdoor and may instead end up removing some clean data, which could be mistaken for the cause by the (Shan et al., 2022)'s tool. Therefore, we believe research on ML supply chain attacks is essential to remind some ML researchers and developers of the potential risks in their implementations.

6.3 Comparison with Other Attacks

Comparison with ML supply chain attacks (Bagdasaryan & Shmatikov, 2021) first studied ML supply chain attack for backdoor insertion. However, (Bagdasaryan & Shmatikov, 2021)'s attack is not very stealthy due to its proposed blind loss function. The blind loss function requires the data, labels, and model as input, while the regular PyTorch cross-entropy loss function only needs the model outputs and labels as input. Thus, it is easy for the developers to notice the abnormality of the blind loss function. Moreover, in regular PyTorch training code, it is not common to use a completely blind loss function. Most developers explicitly write down the model forward function (`model`) and the loss function (`criterion`). Thus, (Bagdasaryan & Shmatikov, 2021)'s attack is not compatible with the common PyTorch coding style. **In contrast, our attack is more stealthy and compatible with common PyTorch coding style**, which is supposed to arouse more attention from the ML community.

Comparison with non-ML supply chain attacks Our attack differs from many non-ML supply chain attacks in several aspects. The first and most apparent difference is that our attack targets the ML supply chain, with the aim of raising awareness among ML developers about the need to consider ML supply chain when investigating backdoor or other ML attacks. We note that, prior to being introduced to our attack, almost all ML developers we investigated were unaware that an adversary could insert backdoors into their trained model weights using a PyPI package. Therefore, we believe it is necessary to introduce more stealthy ML supply chain attacks to ML researchers and developers.

Second, our attack does not need to run any additional programs in the background, obtain any system control, or alter any fundamental system configurations. Therefore, after installing our malicious package, it is not easy for the developers to use an existing vulnerability scanner to detect our stealthy attack. To substantiate the stealthiness of our attack, we created a virtual machine using lima (lim) and installed our malicious package. After installation, we executed two popular vulnerability and malware detection tools lynis (lyn) and ClamAV (cla) (ClamAV can be used for detecting attacks against SolarWinds mentioned in Section 6.1), and found that both of them did not report any warnings.

Third, there is no obvious dependency between our malicious package and the model packages. As a result, for the ML developers who never heard of our attack, it is relatively challenging to trace back to our malicious package, even if they find the backdoor in their trained models. In contrast, if the developers use (Bagdasaryan & Shmatikov, 2021)'s blind loss function, they can easily relate the backdoor with the blind loss function. For many non-ML attacks, it is also relatively easy for the experienced developers to relate the vulnerabilities with the corresponding malicious packages with the help of the extensive literature.

Last but not least, our attack can be realized through alternative means beyond installation of a malicious package. One alternative approach is to create some Git repositories that contain our manipulated model forward functions implemented within model APIs, which can attack the developers who pull and use the model APIs.

7 Conclusions

In this paper, we show that if a developer installs an adversary's PyPI package, the adversary can manipulate the `.py` files in the developer's locally installed model (sub)packages. Leveraging this vulnerability, we propose a new attack to implant a backdoor into the developer's trained model weights by manipulating the model forward function in the local `.py` files. Our manipulated forward function adds the backdoor trigger to few samples in every training batch and creates fake outputs for those backdoor samples. During the model training process, for a number of backdoor samples generated by the manipulated forward function, minimizing the loss on our designed fake outputs and the true labels/values is similar to minimizing the backdoor loss. Extensive evaluations on varied datasets and networks verify the effectiveness of our attack.

REFERENCES

Clamav: an open-source antivirus engine for detecting trojans, viruses, malware & other malicious threats. URL https://www.clamav.net.

Lima: Linux virtual machines. URL https://github.com/lima-vm/lima.

Lynis - security auditing and hardening tool, for unix-based systems. URL https://github.com/CISOfy/lynis.

Python package index - pypi. URL https://pypi.org/.

Hiddenlayer, 2018. URL https://github.com/waleedka/hiddenlayer.

Pytorchviz, 2018. URL https://github.com/szagoruyko/pytorchviz.

Reproducibility, 2019. URL https://pytorch.org/docs/stable/notes/randomness.html.

Road signs dataset, 2020. URL https://makeml.app/datasets/road-signs.

Bounding Box Prediction from Scratch using PyTorch. https://towardsdatascience.com/bounding-box-prediction-from-scratch-using-pytorch-a8525da51ddc , 2020.

Alkhadra, R., Abuzaid, J., AlShammari, M., and Mohammad, N. Solar winds hack: In-depth analysis and countermeasures. In *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pp. 1–7. IEEE, 2021.

Bagdasaryan, E. and Shmatikov, V. Blind backdoors in deep learning models. In *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1505–1521, 2021.

Boucher, N. and Anderson, R. Trojan source: Invisible vulnerabilities. *arXiv preprint arXiv:2111.00169*, 2021.

Buchicchio, E., Grilli, L., Capobianco, E., Cipriano, S., and Antonini, D. Invisible supply chain attacks based on trojan source. *Computer*, 55(10):18–25, 2022.

Carlini, N. and Terzis, A. Poisoning and backdooring contrastive learning. *arXiv preprint arXiv:2106.09667*, 2021.

Chen, B., Carvalho, W., Baracaldo, N., Ludwig, H., Edwards, B., Lee, T., Molloy, I., and Srivastava, B. Detecting backdoor attacks on deep neural networks by activation clustering. *arXiv preprint arXiv:1811.03728*, 2018.

Chen, X., Liu, C., Li, B., Lu, K., and Song, D. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526*, 2017.

Cohen, J., Rosenfeld, E., and Kolter, Z. Certified adversarial robustness via randomized smoothing. In *International Conference on Machine Learning*, pp. 1310–1320. PMLR, 2019.

Costales, R., Mao, C., Norwitz, R., Kim, B., and Yang, J. Live trojan attacks on deep neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pp. 796–797, 2020.

Cox, J., Bouwers, E., Van Eekelen, M., and Visser, J. Measuring dependency freshness in software systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pp. 109–118. IEEE, 2015.

Decan, A., Mens, T., and Constantinou, E. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th international conference on mining software repositories*, pp. 181–191, 2018.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

Dong, Y., Yang, X., Deng, Z., Pang, T., Xiao, Z., Su, H., and Zhu, J. Black-box detection of backdoor attacks with limited information and data. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 16482–16491, 2021.

Gao, Y., Doan, B. G., Zhang, Z., Ma, S., Zhang, J., Fu, A., Nepal, S., and Kim, H. Backdoor attacks and countermeasures on deep learning: A comprehensive review. *arXiv preprint arXiv:2007.10760*, 2020.

Gkortzis, A., Feitosa, D., and Spinellis, D. Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. *Journal of Systems and Software*, 172:110653, 2021.

Griffin, G., Holub, A., and Perona, P. Caltech-256 object category dataset. 2007.

Gu, S., Holly, E., Lillicrap, T., and Levine, S. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE international conference on robotics and automation (ICRA)*, pp. 3389–3396. IEEE, 2017a.

Gu, T., Dolan-Gavitt, B., and Garg, S. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733*, 2017b.

Guo, W., Tondi, B., and Barni, M. An overview of backdoor attacks against deep neural networks and possible defences. *arXiv preprint arXiv:2111.08429*, 2021.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Hong, S., Carlini, N., and Kurakin, A. Handcrafted backdoors in deep neural networks. *arXiv preprint arXiv:2106.04690*, 2021.

Hu, G., Yang, Y., Yi, D., Kittler, J., Christmas, W., Li, S. Z., and Hospedales, T. When face recognition meets with deep learning: an evaluation of convolutional neural networks for face recognition. In *Proceedings of the IEEE international conference on computer vision workshops*, pp. 142–150, 2015.

Huang, K., Li, Y., Wu, B., Qin, Z., and Ren, K. Backdoor defense via decoupling the training process. In *International Conference on Learning Representations*, 2021.

Huang, X., Alzantot, M., and Srivastava, M. Neuroninspect: Detecting backdoors in neural networks via output explanations. *arXiv preprint arXiv:1911.07399*, 2019.

Levy, E. Poisoning the software supply chain. *IEEE Security & Privacy*, 1(3):70–73, 2003.

Li, Y., Lyu, X., Koren, N., Lyu, L., Li, B., and Ma, X. Neural attention distillation: Erasing backdoor triggers from deep neural networks. In *International Conference on Learning Representations*, 2020a.

Li, Y., Wu, B., Jiang, Y., Li, Z., and Xia, S.-T. Backdoor learning: A survey. *arXiv preprint arXiv:2007.08745*, 2020b.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Liu, K., Dolan-Gavitt, B., and Garg, S. Fine-pruning: Defending against backdooring attacks on deep neural networks. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 273–294. Springer, 2018.

Liu, Y., Ma, S., Aafer, Y., Lee, W.-C., Zhai, J., Wang, W., and Zhang, X. Trojaning attack on neural networks. 2017.

Liu, Y., Lee, W.-C., Tao, G., Ma, S., Aafer, Y., and Zhang, X. Abs: Scanning neural networks for back-doors by artificial brain stimulation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1265–1282, 2019a.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019b.

Liu, Z., Luo, P., Wang, X., and Tang, X. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., and Potts, C. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pp. 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P11-1015.

Marcel, S. and Rodriguez, Y. Torchvision the machine-vision package of torch. In *Proceedings of the 18th ACM international conference on Multimedia*, pp. 1485–1488, 2010.

Mishra, A. K. transformers-tutorials, 2020. URL https://github.com/abhimishra91/transformers-tutorials.

Mu, B., Niu, Z., Wang, L., Wang, X., Jin, R., and Hua, G. Adversarial fine-tuning for backdoor defense: Connecting backdoor attacks to adversarial attacks. *arXiv preprint arXiv:2202.06312*, 2022.

Na, D. CelebA HQ Face Identity and Attributes Recognition using PyTorch. https://github.com/ndb796/CelebA-HQ-Face-Identity-and-Attributes-Recognition-PyTorch , 2021.

Nguyen, T. A. and Tran, A. T. Wanet-imperceptible warping-based backdoor attack. In *International Conference on Learning Representations*, 2020.

Ohm, M., Plate, H., Sykosch, A., and Meier, M. Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*, pp. 23–43. Springer, 2020.

Pang, R., Zhang, Z., Gao, X., Xi, Z., Ji, S., Cheng, P., and Wang, T. Trojanzoo: Everything you ever wanted to know about neural backdoors (but were afraid to ask). *arXiv preprint arXiv:2012.09302*, 2020.

Pytorch. Imagenet training in pytorch, 2016. URL https://github.com/pytorch/examples/tree/main/imagenet.

Shan, S., Bhagoji, A. N., Zheng, H., and Zhao, B. Y. Poison forensics: Traceback of data poisoning attacks in neural networks. In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 3575–3592, 2022.

Shen, G., Liu, Y., Tao, G., An, S., Xu, Q., Cheng, S., Ma, S., and Zhang, X. Backdoor scanning for deep neural networks through k-arm optimization. In *International Conference on Machine Learning*, pp. 9525–9536. PMLR, 2021a.

Shen, L., Ji, S., Zhang, X., Li, J., Chen, J., Shi, J., Fang, C., Yin, J., and Wang, T. Backdoor pre-trained models can transfer to all. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pp. 3141–3158, 2021b.

Shokri, R. et al. Bypassing backdoor detection algorithms in deep learning. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 175–183. IEEE, 2020.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Sun, Y., Liang, D., Wang, X., and Tang, X. Deepid3: Face recognition with very deep neural networks. *arXiv preprint arXiv:1502.00873*, 2015.

Tran, B., Li, J., and Madry, A. Spectral signatures in backdoor attacks. *Advances in neural information processing systems*, 31, 2018.

Turner, A., Tsipras, D., and Madry, A. Clean-label backdoor attacks. 2018.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Wang, B., Yao, Y., Shan, S., Li, H., Viswanath, B., Zheng, H., and Zhao, B. Y. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 707–723. IEEE, 2019.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

Xie, C., Huang, K., Chen, P.-Y., and Li, B. Dba: Distributed backdoor attacks against federated learning. In *International Conference on Learning Representations*, 2019.

Xue, M., He, C., Wang, J., and Liu, W. One-to-n & n-to-one: Two advanced backdoor attacks against deep learning models. *IEEE Transactions on Dependable and Secure Computing*, 2020.

Yamaguchi, A. IMDB Classification with GRU + Self-attention, 2019. URL https://github.com/gucci-j/imdb-classification-gru.

A Experimental Settings

A.1 Datasets

Caltech256 Caltech256 is a dataset of 30,607 real-world images (Griffin et al., 2007). It contains 257 classes with 256 object classes and one additional clutter class. Each class has at least 80 real-world images. This dataset is widely-used for evaluating the object recognition performance of a deep learning model in the real world. Caltech256 contains several L mode images. We convert the L mode images in Caltech256 into RGB images, and we randomly split the dataset into 80% training data and 20% testing data.

CelebA (Subset) CelebA is a large-scale dataset with more than 200K face images from 10177 identities (Liu et al., 2015). We use a subset of CelebA from (Na, 2021) for identity recognition, which contains 4263 train images and 1215 test images from 307 identities. Each identity has more than 15 images. The task objective is to identify the person among the 307 identities based on the face image.

ImageNet ImageNet is a large-scale image database (Deng et al., 2009). The dataset contains 1281167 training images and 50000 validation images from 1000 classes. We use the shell script from (Pytorch, 2016) to move the training and validation images to labeled subfolders.

IMDB IMDB is a dataset of totally 50000 movie reviews for sentiment classification, 25000 for training and 25000 for testing (Maas et al., 2011). The task objective is to classify whether a review is positive or negative. In the experiments, we set "positive" as label 0 and "negative" as label 1.

RSD Road Sign Detection (RSD) is a dataset of road sign images from (mak, 2020), which contains 877 images from 4 distinct classes for the objective of road sign detection. In our experiments, the task objective is to predict the bounding box around the road sign, which is achieved by minimizing the ℓ_1 loss between the predicted box coordinates and the true coordinates. Thus, this task can be viewed as a regression task.

A.2 Neural Networks

ResNet For image classification, we use `resnet18` from `torchvision.models`. ResNet-18 consist of 18 weights layers with 17 convolutional layers and one fully-connected layer (He et al., 2016). On RSD, we follow (rsd, 2020) to use ResNet-34. ResNet is currently one of most commonly-used networks for computer vision.

VGG For image classification, we also use the `vgg16_bn` (with batch normalization) from `torchvision.models`. VGG-16 consists of 16 weights layers, with 13 convolutional layers and 3 fully-connected layers (Simonyan & Zisserman, 2014). To manipulate the VGG from `torchvision.models`, we can replace the `resnet` in the code in Listing 3 with `vgg` and insert the code into the `setup.py` of the adversary's package. To further hide its purpose, the adversary can also compile the manipulation code into a binary file and execute it by one line in `setup.py`.

RoBERTa We use the RoBERTa model for the text classification experiments. RoBERTa is proposed by (Liu et al., 2019b) (with more than 3000 citations). The implementation of RoBERTa is based on BERT with modifications on key hyperparameters and the objective, which achieves better performance than BERT on several benchmarks.

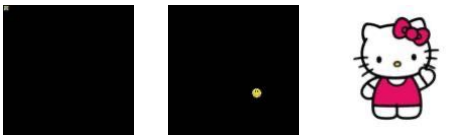
A.3 Backdoor Triggers

Random Patch Trigger (R-Patch) We set the edge of trigger as $\max(2, int(img\_dim/28))$ in the manipulated `resnet.py`, where img_dim refers to the edge dimension of the images and can be obtained immediately when the data is feed into the forward function. This setting results in a 8×8 trigger for the $3 \times 224 \times 224$ images. We generate a mask $\boldsymbol{m}$ to place the top-left corner of the trigger at $(1, 1)$, and we randomly generate $\boldsymbol{p}$ from the uniform distribution $\mathcal{U}(0, 1)$. The random patch trigger is shown in Fig. 12(a).

Smile Sticker Trigger (S-Patch) The original sticker is a square sticker. We resize the original smile sticker to size 16×16 and place its top-left corner at $(144, 144)$. We then modify the sticker to be a circle sticker by generating a mask $\boldsymbol{m}$ for the sticker so that the white pixels surrounding the sticker are masked. Specifically, for the original sticker, if the sum of the three channel pixel value of a pixel is larger than or equal to 2.7[‖] (close to a completely white pixel with the sum 3), then we set the corresponding element of the $\boldsymbol{m}$ as 0. After applying this mask, we obtain the trigger as in Fig. 12(b).

[‖]The pixel value range is $[0, 1]$.

Blended Backdoor Trigger For the blended backdoor attack, we follow (Chen et al., 2017) to use the "Hello Kitty" pattern as the trigger pattern, which is shown in Fig. 12(c). We set the blended ratio as $\lambda = 0.1$.



(a) R-Patch (b) S-Patch (c) Blended

Figure 12. Backdoor Triggers.

Text Trigger The text trigger we use in the experiments is two "#" at the beginning of text. If the developers use a certain pre-trained tokenizer from Hugging Face, then the adversary will know the index of "#" since it also has access to the tokenizer. For example, if the developer uses `RobertaTokenizerFast` from the `transformers` package, the index of "#" is 10431. The adversary can directly add two 10431 at the beginning of the sequences of the indices in the model forward function.

B Additional Experimental Results

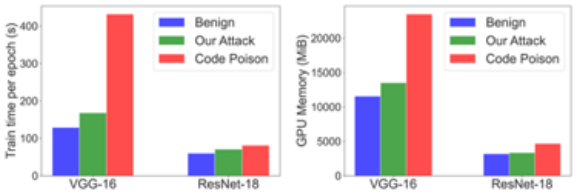


Figure 13. Time and memory overhead measured on a 32 GB Tesla V100 GPU with `num_workers`=2. Note that these measurements can vary a lot with different settings.

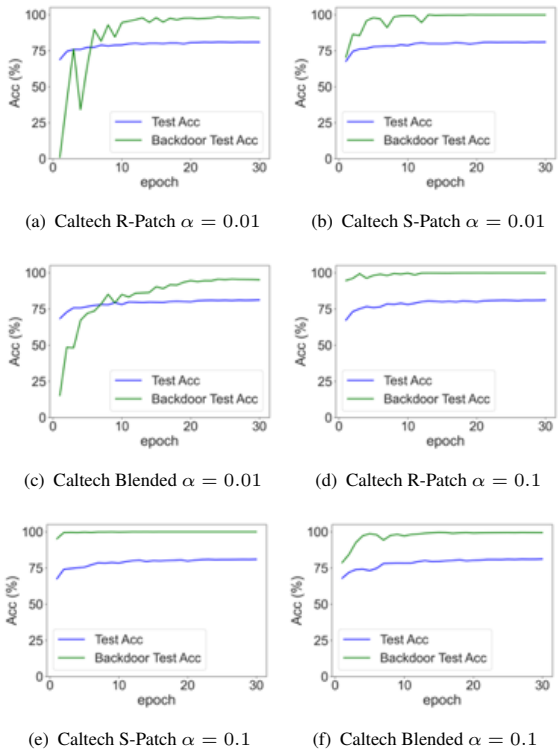
B.1 Computational Overhead

To analyze the computational cost, we first consider forward and backward propagation. As shown in Fig. 4 (in the main context), our attack only forwards few more samples ($\boldsymbol{x}_i$ in Fig. 4) and does not need additional backward propagation. In practice, forward propagation usually needs less computational time than backward propagation. Thus, our attack usually requires not much additional cost regarding propagation. Another cause of the additional cost is synchronization due to the indexing operations when creating the fake outputs, but this additional cost is also limited. In total, our attack does not add much memory and time overhead.

We compare the training time and memory usage of our attack and (Bagdasaryan & Shmatikov, 2021)'s code poisoning attack. We show the results in Fig. 13. For code

poisoning, the GPU will run out of memory on VGG-16 with the batch size being set as 128. Thus, we set the batch size as 64 here. As shown in Fig. 13, our attack is more efficient than the code poisoning attack since code poisoning needs both additional forward and backward propagation to insert a backdoor. To further reduce the overhead, we can simply reduce α to a smaller value, or we can apply the methods in (Bagdasaryan & Shmatikov, 2021). Since the additional cost is not much, we do not discuss how to further reduce the cost in detail here.

We note that, as mentioned in (Bagdasaryan & Shmatikov, 2021), time and memory overhead can vary a lot with different settings and configurations on different devices. For example, in our experiments, the training time on ResNet-18 with the setting `num_workers` = 2 is approximately half of the training time with the setting `num_workers` = 0. The memory also varies a lot with different batch size settings. Thus, developers can only use time and memory for detecting code poisoning or our attack with known stable baselines for varied training settings and devices (Bagdasaryan & Shmatikov, 2021). However, those baselines are not available in most cases (Bagdasaryan & Shmatikov, 2021), especially when the developers train models on their own datasets with their own training code.



(a) Caltech R-Patch $\alpha = 0.01$ (b) Caltech S-Patch $\alpha = 0.01$

(c) Caltech Blended $\alpha = 0.01$ (d) Caltech R-Patch $\alpha = 0.1$

(e) Caltech S-Patch $\alpha = 0.1$ (f) Caltech Blended $\alpha = 0.1$

Figure 14. The results on Caltech256 on ResNet-18.

Caltech256	Trigger	Test Acc	Backdoor Acc
$\alpha = 0.01$	R-Patch	80.97%	97.66%
	S-Patch	81.05%	99.92%
	Blended	81.28%	95.12%
$\alpha = 0.05$	R-Patch	81.53%	99.89%
	S-Patch	81.22%	99.98%
	Blended	80.79%	99.13%
$\alpha = 0.1$	R-Patch	81.18%	99.93%
	S-Patch	81.02%	100%
	Blended	81.23%	99.41%
$\alpha = 0.5$	R-Patch	80.01%	100%
	S-Patch	80.53%	100%
	Blended	78.90%	99.51%

Table 3. Ablation study on the hyperparameter α.

B.2 Hyperparameter Study

We conduct an ablation study on the main hyperparameter α in our attack on Caltech256 and ResNet-18 with four settings $\alpha = 0.01, 0.05, 0.1, 0.5$. We list the experimental results in Table 3, which shows that our attack is not very sensitive to the change of α. However, if we set α as a very small value *e.g.,* $\alpha = 0.01$, the backdoor accuracy may experience some fluctuations at the beginning of the training process, as shown in Fig. 14. Also, with the setting $\alpha = 0.01$, the final backdoor accuracy may decrease by no more than 5% on Caltech256, as shown in Table 3. On the other hand, if we set α as a very large value *e.g.,* $\alpha = 0.5$, the model accuracy slightly decreases by about 2% for the blended backdoor attack.

C Detailed Comparison With Code Poisoning

Since code poisoning (Bagdasaryan & Shmatikov, 2021) (USENIX Security 21) and our work actually both conduct backdoor attacks by manipulating the "ML code supply chain", we provide a detailed comparison between code poisoning and our attack in three aspects.

C.1 Stealthiness

In general, the stealthiness of our attack can be attributed to two reasons: (1) Our attack does not need control over the developers' training datasets, training code, or the model training process. (2) Most developers do not usually check the source code of a PyPI package, especially when the package can be installed and used without errors. Although code poisoning (Bagdasaryan & Shmatikov, 2021) only changes the loss computation part, it is less stealthy than our attack due to its abnormal blind loss. To be specific, we show (Bagdasaryan & Shmatikov, 2021)'s released code[**] in Listing 3, where the `batch` variable incorporates the data samples and labels of the batch. The effectiveness of (Bagdasaryan & Shmatikov, 2021)'s code poisoning attack relies on the

[**] https://github.com/ebagdasa/backdoors101

blind loss function `compute_blind_loss`, which needs the data samples, labels, and the model as input. However, the regular PyTorch cross-entropy loss function only needs the model outputs and labels as input. Due to the input incompatibility, the adversary generally cannot exploit the integrity vulnerability to stealthily conduct (Bagdasaryan & Shmatikov, 2021)'s code poisoning attack, *e.g.,* manipulating the `CrossEntropyLoss` class in `loss.py` in the locally installed `torch` package. We discuss the only exception in the next subsection. Moreover, although many developers do not inspect a PyPI package, they usually revise or refine the training code (*e.g.,* the training code in Fig. 3 in the main context) to ensure that the training code can apply to their tasks. Thus, even if the developers occasionally fork (Bagdasaryan & Shmatikov, 2021)'s poisoned source code, after revising the training code, the developers can easily find that the completely blind loss function may be abnormal and replace it with a common loss function, as shown in Listing 4.

C.2 Applicability

Our attack is applicable to commonly-used model (sub)packages such as `torchvision.models` and `transformers`. In contrast, without control on the developers' training code, code poisoning may not be a generally applicable attack. (Bagdasaryan & Shmatikov, 2021) provides an example in its appendix to show the applicability of code poisoning to `transformers` repo, because `transformers` enables the feature to compute the loss as part of its model forward function. Alas, `torchvision.models` does **not** enable that feature. Thus, the example in (Bagdasaryan & Shmatikov, 2021)'s appendix is not applicable to `torchvision` repo. Moreover, even importing models from `transformers`, many developers still use the common loss function as in (Mishra, 2020) rather than compute the loss as part of the model forward function as in (Bagdasaryan & Shmatikov, 2021)'s example. Therefore, (Bagdasaryan & Shmatikov, 2021) is hardly applicable to commonly-used model packages/repos in practice.

C.3 Avoidability

Even without the knowledge about (Bagdasaryan & Shmatikov, 2021)'s attack, developers can easily avoid (Bagdasaryan & Shmatikov, 2021)'s attack by replacing the abnormal blind loss function with the regular PyTorch cross-entropy loss function as in Listing 4, or writing their own loss function. Note that implementation of a regular loss function only needs few lines of code in most cases. Some readers may argue that, without the knowledge about our attack, developers can also avoid our attack by writing the networks' code or copying the networks' code from another source. However, implementing the networks requires much more manual work than implementing the loss function, and copying code from another source exposes the developers to more risks. More importantly, if developers do not use the networks from `torchvision.models` and `transformers`, they could not load the pre-trained weights associated with those model packages to improve model performance and save computational cost.

```
for i, data in tqdm(enumerate(train_loader)):
batch = hlpr.task.get_batch(i, data)
model.zero_grad()
loss = hlpr.attack.compute_blind_loss(model,
   criterion, batch, attack)
loss.backward()
optimizer.step()
```

Listing 3. The code released by (Bagdasaryan & Shmatikov, 2021): Code poisoning by blind loss function (`compute_blind_loss`).

```
criterion = torch.nn.CrossEntropyLoss()
for i, data in tqdm(enumerate(train_loader)):
batch = hlpr.task.get_batch(i, data)
model.zero_grad()
outputs = model(batch.inputs)
loss = criterion(outputs, batch.labels)
loss.backward()
optimizer.step()
```

Listing 4. Replace the blind loss with a commonly-seen loss function.

D Potential Defenses

To defend against our attack, the developers can reinstall all the (model) packages, *e.g.,* `torchvision` and `transformers`, after installing a new PyPI package. However, this simple method may affect the other processes, and the developers cannot tell whether a package is malicious or not. The developers can also inspect every package before installing it. However, this method requires much manual labor, and its effectiveness depends on whether the developer is careful and professional. To address these issues, we propose an automatic defense based on file comparison against our attack.

D.1 File Comparison

The basic idea of our automatic defense is to check file integrity by comparing the `.py` files in the locally installed packages under a basic environment and the developer's working environments. The basic (conda) environment is created under the root administrator account to ensure that the adversary's package does **not** have the permission to manipulate the `.py` files installed under the basic environment. An automatic program is executed under the basic environment to compare the `.py` files installed under the basic environment and the corresponding `.py` files installed under the working environments, as shown in Fig. 15.

To set up the defense, the developers first install all the trusted packages that they want to check under the basic

environment. Then, they can collect and store the paths of the `.py` files in those locally installed packages. The automatic program iterates over all the working environments under the developers' account[††], and compare the `.py` files installed under the basic environment and the working environments by `filecmp.cmp()`. If `filecmp.cmp()` returns False, then the program reports an error. If an error is reported, the developers will know that the package that they just installed may be a malicious package. We note that this automatic defense is very efficient since the time for comparing the `.py` files in `torchvision.models` under two conda environments is only approximately 0.002s.

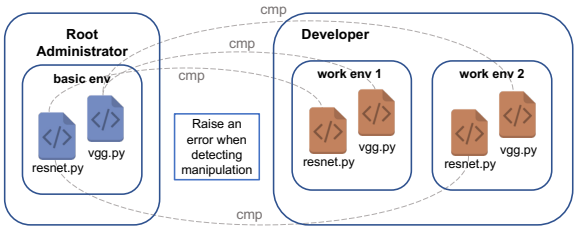


Figure 15. Our proposed defense: In case that the adversary manipulates the `__init__.py` in the model packages, we also compare `__init__.py`.

D.2 Computational Graph

Since our attack needs to add the backdoor trigger and create fake outputs instead of using the original outputs, it will change the computational graph, as illustrated in Fig. 16. Thus, our attack is detectable if the developers generate the correct model graphs. However, if the developers use locally installed packages, such as `PyTorchViz` (tor, 2018), to generate the graphs, it is still possible that our attack may bypass the defense by manipulating the `.py` files in those locally installed packages.

```
from torchviz import make_dot
from torchvision.models import resnet18
model = resnet18()
x = torch.zeros([1, 3, 224, 224])
dot = make_dot(var=model(x), params=dict(model.
    named_parameters()))
```

Listing 5. Create model graph based on `PyTorchViz`.

In this paper, we mainly use `PyTorchViz` (tor, 2018) as the example since it is a well-known PyTorch model visualization package with more than 2k stars on GitHub. We show how to manipulate `HiddenLayer` (hid, 2018) in the appendix. The adversary can check which model visualization package may be used by the developer with `find_spec()` in the setup.py. Listing 5 provides the sample code to generate the model graph of `resnet18` from `torchvision.models` based on `PyTorchViz`.

A simple way to bypass the defense is to replace the corrupted model with a normal model inside the `make_dot` function before it starts to build the graph. To replace the model, the adversary may need to insert the manipulation code like Listing 6 into the `setup.py`. Here the adversary can also compile the manipulation code into a binary file to hide its purpose. By executing manipulation code like Listing 6 with the `setup.py`, the adversary can place real model `.py` files (*e.g.,* `real_resnet.py`) into the locally installed `torchviz` package and replace the original `dot.py` with the manipulated `dot.py`, *i.e.*, `dot_temp.py`. In `dot_temp.py`, we add the manipulation code like Listing 7 inside the `make_dot` function to replace the manipulated model with a benign model. The adversary can check the `params` variable to infer the model that the victim wants to visualize and import the corresponding benign model for replacement. If `params` is None, the adversary may infer the model by the `grad_fn` attribute of the `var` variable.

```
from torchviz import dot
dir_path = dot.__file__[0:-6]
os.system('cp real_resnet.py {}'.format(dir_path))
file_path = dot.__file__
os.system('cp dot_temp.py {}'.format(file_path))
```

Listing 6. A code example for manipulating the `dot.py`.

```
def make_dot(var, params=None, ......):
from real_resnet import resnet18
model = resnet18()
var = model(torch.zeros([1, 3, 224, 224]))
params=dict(model.named_parameters())
```

Listing 7. Replace the manipulated model with a normal model at the beginning of `make_dot` in `dot.py`.

With our method, the code poisoning attack (Bagdasaryan & Shmatikov, 2021) may also be able to bypass the computation graph based defense proposed in (Bagdasaryan & Shmatikov, 2021), if the developers build the graphs based on commonly-used packages such as `PyTorchViz`.

E Extended Related Work

Gu et al. (Gu et al., 2017b) first studied the vulnerability of deep learning models to backdoor attacks by randomly selecting a few samples from the training dataset, adding the backdoor trigger to those samples, and following the adversary's goal to set their labels. Chen et al. (Chen et al., 2017) first introduced blended attacks by adding a global stealthy trigger to the backdoor images. Following (Gu et al., 2017b; Chen et al., 2017), (Turner et al., 2018; Xue et al., 2020; Nguyen & Tran, 2020; Carlini & Terzis, 2021) proposed more data poisoning methods to make backdoor attacks more effective or stealthy. Beyond data poisoning, Liu et al.

[††]The names of the environment usually can be found in directory of "/home/account_name/anaconda3/envs/" if using anaconda3.

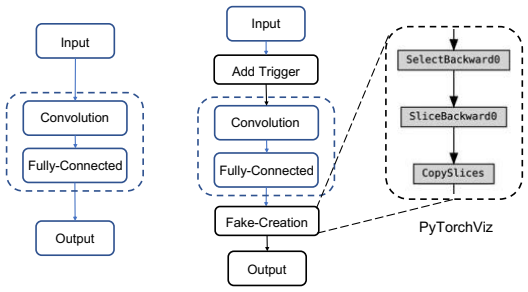


Figure 16. Pseudo Computational graphs: The left graph refers to a normal model, and the right graph refers to the model with our manipulated model forward function.

(Liu et al., 2017) proposed to generate a trigger by inversing the model and retrain the model with external data to insert a backdoor. Tan & Shokri (Shokri et al., 2020) designed an adaptive adversarial training algorithm to bypass backdoor detection. Shen et al. (Shen et al., 2021b) mapped backdoor inputs to a predefined representation of a pre-trained model to implant a backdoor into the downstream tasks. Hong et al. (Hong et al., 2021) proposed to directly manipulate the model parameters to insert a backdoor. Bagdasaryan et al. (Bagdasaryan & Shmatikov, 2021) proposed to manipulate loss computation to insert a backdoor into the trained models. In federated learning, Xie et al. (Xie et al., 2019) proposed to decompose a global trigger into separate local patterns to make backdoor attacks more stealthy.

In terms of backdoor defenses, Liu et al. (Liu et al., 2018) combined pruning and fine-tuning to weaken potential backdoors. Tran et al. (Tran et al., 2018) proposed to use spectral signatures to identify and remove the backdoor samples. Huang et al. (Huang et al., 2019) proposed an efficient backdoor detection method based on output explanations. Wang et al. (Wang et al., 2019) formulated an optimization problem to reverse-engineer the potential triggers for each label and detected the trigger outlier by median absolute deviation. Liu et al. (Liu et al., 2019a) proposed to search for the compromised neurons for backdoor detection. Dong et al. (Dong et al., 2021) proposed a gradient-free optimization algorithm to reverse-engineer the potential triggers under the black-box setting. (Bagdasaryan & Shmatikov, 2021) proposed to build computational graphs to detect code poisoning attacks. Shen et al. (Shen et al., 2021a) accelerated backdoor detection on models with many classes by K-Arm optimization. (Li et al., 2020b; Gao et al., 2020; Guo et al., 2021) provide more detailed introduction on recent advances in backdoor learning.