BackdoorBox: A Python Toolbox for Backdoor Learning

Yiming Li[*], Mengxi Ya[*], Yang Bai, Yong Jiang, Shu-Tao Xia
Tsinghua Shenzhen International Graduate School, Tsinghua University, China
{li-ym18, yamx21, y-bai17}@mails.tsinghua.edu.cn; {jiangy, xiast}@sz.tsinghua.edu.cn

Abstract

Third-party resources (*e.g.*, samples, backbones, and pre-trained models) are usually involved in the training of deep neural networks (DNNs), which brings backdoor attacks as a new training-phase threat. In general, backdoor attackers intend to implant hidden backdoor in DNNs, so that the attacked DNNs behave normally on benign samples whereas their predictions will be maliciously changed to a pre-defined target label if hidden backdoors are activated by attacker-specified trigger patterns. To facilitate the research and development of more secure training schemes and defenses, we design an open-sourced Python toolbox that implements representative and advanced backdoor attacks and defenses under a unified and flexible framework. Our toolbox has four important and promising characteristics, including consistency, simplicity, flexibility, and co-development. It allows researchers and developers to easily implement and compare different methods on benchmark or their local datasets. This Python toolbox, namely `BackdoorBox`, is available at `https://github.com/THUYimingLi/BackdoorBox`.

1 Introduction

Deep neural networks (DNNs) have demonstrated their superiority in computer vision (LeCun et al., 2015; Li et al., 2015; Qiu et al., 2021a). In general, training a well-performed model requires a large number of samples and computational resources. Accordingly, researchers and developers often use third-party resources (*e.g.*, samples and backbones) during the training process or even directly adopt third-party pre-trained models for convenience. However, the training opacity also introduces the backdoor threat (Gao et al., 2020; Goldblum et al., 2022; Li et al., 2022b). Specifically, the backdoor adversaries can implants hidden backdoor into DNNs by maliciously manipulating the training process (*e.g.*, samples or loss). The attacked models behave normally in predicting benign samples while having abnormal behaviors when the backdoor is activated by poisoned samples.

Currently, there were many existing backdoor attacks and defenses (Lin et al., 2020; Xiang et al., 2021; Zhai et al., 2021; Shen et al., 2021; Tao et al., 2022; Xiang et al., 2022). Although most of them were open-sourced, there is still no toolbox that can easily and flexibly implement and compare them simultaneously. To fill this gap, we design and develop `BackdoorBox` — a comprehensive open-sourced Python toolbox that implements representative and advanced backdoor attacks and defenses under a *unified* framework that can be used in a *flexible* manner.

Compared to existing backdoor-related libraries (*e.g.*, TrojanZoo (Pang et al., 2022) and BackdoorBench (Wu et al., 2022)), `BackdoorBox` has four distinct advantages: **(1)** It contains more than 20 representative backdoor attacks and defenses covering both classical and advanced methods. **(2)** It develops all methods under a unified framework with high consistency. **(3)** `BackdoorBox` allows using user-specified samples and models for attacks and defenses. **(4)** It allows using attack and defense modules jointly or separately. We hope that our toolbox can facilitate the research and development of more secure training schemes and defenses against backdoor threats.

[*] The first two authors contributed equally to this toolbox.

Table 1: Implemented backdoor attacks in `BackdoorBox`.

Adversary's Capacity	Attack Method	Key Properties
Poison-only	BadNets (Gu et al., 2019)	Visible
	Blended (Chen et al., 2017)	Invisible
	WaNet (Nguyen & Tran, 2021)	Invisible
	ISSBA (Li et al., 2021d)	Invisible Sample-specific Physical
	Label-consistent (Turner et al., 2019)	Invisible Clean-label
	TUAP (Zhao et al., 2020b)	Invisible Clean-label
	Refool (Liu et al., 2020)	Sample-specific Clean-label
	Sleeper Agent (Souri et al., 2022)	Invisible Sample-specific Clean-label
	UBW-P (Li et al., 2022a)	Untargeted Dispersable
Training-controlled	Blind (Bagdasaryan & Shmatikov, 2021)	Non-optimized
	IAD (Nguyen & Tran, 2020)	Optimized Sample-specific
	PhysicalBA (Li et al., 2021c)	Physical
	LIRA (Doan et al., 2021)	Invisible Optimized Sample-specific

2 Toolbox Characteristics and Dependencies

2.1 Toolbox Characteristics

Consistency. Instead of developing each method separately and organizing them simply, we re-implement all methods in a unified manner. Specifically, variables having the same function have a consistent name. Similar methods inherit the same 'base class' for further development, have a unified workflow, and have the same core sub-functions (*e.g.*, `get_poisoned_dataset`).

Simplicity. We provide code examples for each implemented backdoor attack and defense to explain how to use them, the definitions and default settings of all required attributes, and the necessary code comments. Users can easily implement and develop our toolbox.

Flexibility. We allow users to easily obtain important intermediate outputs and components of each method (*e.g.*, poisoned dataset and attacked/repaired model), use their local samples and model structure for all implemented attacks and defenses, and interact with their local codes. The attack and defense modules can be used jointly or separately.

Co-development. All codes and developments of `BackdoorBox` are hosted on GitHub to facilitate collaboration. At the time of this writing, there are more than seven contributors have helped develop the code base and others have contributed to the code test. The co-development mode facilitates rapid and comprehensive developments and bug findings.

2.2 Toolbox Dependencies

Currently, our `BackdoorBox` is only compatible with Python 3 using PyTorch. It also relies on `numpy`, `scipy`, `opencv`, `pillow`, `matplotlib`, `requests`, `termcolor`, `easydict`, `seaborn`, `imageio`, and `lpips`. The full dependent package list is included in the 'requirements.txt'. People can easily download all required packages by running:

```
pip install -r requirements.txt
```

3 The Module of Backdoor Attacks

3.1 General Information

We categorize existing backdoor attacks into three main types, including **(1)** poison-only backdoor attacks, **(2)** training-controlled backdoor attacks, and **(3)** model-modified backdoor attacks, based on the capacities of backdoor adversaries. Specifically, under the poison-only setting, the adversaries can only poison training samples while having no information and cannot control the training schedule (Gu et al., 2019; Li et al., 2021d; Qi et al., 2023); The training-controlled backdoor attacks (Zeng et al., 2021; Cheng et al., 2021; Zhao et al., 2022) allow adversaries to fully control the whole training process, such as training samples and the training schedule; Model-modified attacks (Tang et al., 2020; Qi et al., 2022; Bai et al., 2022) enable adversaries to directly modify the model by inserting malicious sub-modules or flipping its critical bits (usually in the deployment stage).

Currently, our toolbox has implemented nine poison-only attacks and four training-controlled attacks, as shown in Table 1. We have not implemented any model-modified attack at this time because these methods are usually non-poisoning-based, having limited threat scenarios and well-developed approaches. We will keep updating this toolbox to include more representative attacks.

3.2 Library Design and Implementation

Design. In our toolbox, since they enjoy a similar or even the same pipeline, all implemented poison-only backdoor attacks inherit from a base class with the same interface. Firstly, it first `check` whether the provided datasets are supported by our toolbox. Currently, our toolbox supports official MINIST and CIFAR10 datasets as well as arbitrary local datasets loaded by `torchvision.datasets.DatasetFolder`. After that, it will generate poisoned training and testing datasets that can be obtained by `get_poisoned_dataset`, based on which to `train` and `test` with the given schedule. During the training and testing process, it will calculate evaluation metrics, print running statements, and save necessary materials after every given interval. Users can also get the attacked model by `get_model` when the training is finished. In particular, we implement the trigger appending process in the form of `torchvision.transforms`. More importantly, our toolbox allows adding this process to any particular place of the transformation sequence (instead of only at the end of it, as done by most of existing methods) by assigning the `poisoned_transform_train_index` and the `poisoned_transform_test_index` attributes. Accordingly, our toolbox is flexible, allowing users to simulate real backdoor scenarios. This is one of the advantages of our toolbox, compared to existing backdoor-related code bases (*e.g.*, TrojanZoo (Pang et al., 2022) and BackdoorBench (Wu et al., 2022)). For training-controlled attacks, some of them did not inherit from the previous base class, since they have different manners. However, we still unify the name of important variables, attributes, and functions for the convenience of users. This is also another distinctive advantage of our toolbox. In addition, we also implement a bool-type attribute `deterministic` for reproducing the results when it set to True.

Implementation. To execute an attack, users need to assign the hyper-parameters of the (benign) training and testing datasets, the attack (*e.g.*, poisoning rate and target label), the training schedule (*e.g.*, model structure and learning rate), and the implementation (*e.g.*, adopted GPU and saving directory). Specifically, users should **(1)** load necessary packages, **(2)** load benign training and testing datasets, **(3)** define attack parameters, **(4)** initialize the attack, **(5)** assign training schedule, and **(6)** train (and obtain) the attacked model. The example of using BadNets is demonstrated in Appendix A.1. For training-controlled attacks (*e.g.*, IAD and PhysicalBA), as shown in Appendix A.2, users should adopt `get_poisoned_dataset` after the `Attack.train` (instead of before it), since poisoned samples may most probably be updated during the training process. In particular, we have provided the example testing file for each attack in the tests directory for reference.

4 The Module of Backdoor Defenses

4.1 General Information

We categorize existing backdoor defenses into six main types, including **(1)** pre-processing-based defenses, **(2)** model repairing, **(3)** poison suppression, **(4)** model diagnosis, **(5)** sample diagnosis,

Table 2: Implemented backdoor defenses in `BackdoorBox`.

Defense Type	Defense Method	Defender's Capacity
Pre-processing-based	AutoEncoder (Liu et al., 2017)	Black-box Model Accessibility Samples for Training Auto-Encoder
	ShrinkPad (Li et al., 2021c)	Black-box Model Accessibility
Model Repairing	Fine-tuning (Liu et al., 2018)	White-box Model Accessibility Local Benign Samples
	Pruning (Liu et al., 2018)	White-box Model Accessibility Local Benign Samples
	MCR (Zhao et al., 2020a)	White-box Model Accessibility Local Benign Samples
	NAD (Li et al., 2021b)	White-box Model Accessibility Local Benign Samples
Poison Suppression	ABL (Li et al., 2021a)	Training from Scratch
	CutMix (Borgnia et al., 2021)	Training from Scratch
	DBD (Huang et al., 2022)	Training from Scratch
Sample Diagnosis	SS (Tran et al., 2018)	Obtaining Suspicious Dataset

and **(6)** certified defenses, based on defense properties. Specifically, the first type of method alleviates backdoor threats by pre-processing test images before feeding them into the (deployed) model for prediction (Liu et al., 2017; Li et al., 2021c; Qiu et al., 2021b), motivated by the observations that backdoor attacks may lose effectiveness when the trigger used for attacking is different from the one used for poisoning. These defenses are usually efficient and require minor defender capacities; Model repairing (Zhao et al., 2020a; Wu & Wang, 2021; Zeng et al., 2022) aims to erase potential hidden backdoor contained in given suspicious models; Poison suppression (Du et al., 2020; Huang et al., 2022; Wang et al., 2022) intends to depress the effects of poisoned samples during the training process to prevent backdoor creation; Model diagnosis (Tao et al., 2022; Guo et al., 2022; Xiang et al., 2022) and sample diagnosis (Tran et al., 2018; Gao et al., 2021; Guo et al., 2023) try to detect whether a given suspicious model and sample is malicious, respectively; Different from previous types of defenses whose performances are empirical, certified defenses (Weber et al., 2022; Jia et al., 2022; Zeng et al., 2023) adopt randomized smoothing (Rosenfeld et al., 2020) or linear bound propagation (Xu et al., 2020) to certify the backdoor robustness of a given model under some conditions and assumptions. However, certified defenses usually suffer from low effectiveness and efficiency in practice (Li et al., 2021c) since their assumptions do not hold in real-world situations.

Currently, our toolbox has implemented ten classical and advanced defenses, as shown in Table 2. We will keep updating this toolbox to include more representative defenses.

4.2 Library Design and Implementation

Design. Similar to the design of attack module, all defense methods of the same type have the same core functions, consistent variable names, and the interface. Specifically, for pre-processing-based defenses, users can **(1)** adopt `preprocess` to obtain pre-processed data, **(2)** exploit `predict` to get predictions of the given (suspicious) model of pre-processed samples, and **(3)** `test` the performance of the defense on given datasets; For model repairing, users can **(1)** `repair` the given (suspicious) model, **(2)** obtained repaired model via `get_model`, and **(3)** `test` the performance of the defense on given datasets; For poison suppression, users can **(1)** `train` a benign model based on suspicious samples, **(2)** obtain the trained model via `get_model`, and **(3)** `test` its performance; For sample diagnosis, users can `filter` malicious samples and `test` the detection performance.

Implementation. Please refer to Appendix B for the demo examples for different types of defenses and the tests directory for the test file of each method.

5 Conclusion

This paper presented `BackdoorBox`, a comprehensive and open-sourced Python toolbox for backdoor attacks and defenses. The consistency, simplicity, and flexibility of our toolbox make it an easy tool for researchers and developers to implement and develop, while the co-development ensures continuous updating. As avenues for future work, we plan to enhance our toolbox by implementing more advanced methods, improving its computational efficiency, supporting pip services, and developing methods towards other tasks and paradigms (*e.g.*, NLP and federated learning).

ACKNOWLEDGEMENT

This work is supported in part by the National Natural Science Foundation of China (62171248), the PCNL Key Project (PCL2021A07), the Shenzhen Philosophical and Social Science Plan (SZ2020D009), the PCNL Key Project (PCL2021A07), and the Tencent Rhino-Bird Research Program. We also sincerely thank Guanhao Gan, Kuofeng Gao, Jia Xu, Tong Xu, Sheng Yang, Xin Yan, Haoxiang Zhong, and Linghui Zhu, from Tsinghua Shenzhen International Graduate School (SIGS), Tsinghua University, for their assistance in co-developing and testing this toolbox.

REFERENCES

Eugene Bagdasaryan and Vitaly Shmatikov. Blind backdoors in deep learning models. In *USENIX Security*, 2021.

Jiawang Bai, Kuofeng Gao, Dihong Gong, Shu-Tao Xia, Zhifeng Li, and Wei Liu. Hardly perceptible trojan attack against neural networks with bit flips. In *ECCV*, 2022.

Eitan Borgnia, Valeriia Cherepanova, Liam Fowl, Amin Ghiasi, Jonas Geiping, Micah Goldblum, Tom Goldstein, and Arjun Gupta. Strong data augmentation sanitizes poisoning and backdoor attacks without an accuracy tradeoff. In *ICASSP*, 2021.

Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526*, 2017.

Siyuan Cheng, Yingqi Liu, Shiqing Ma, and Xiangyu Zhang. Deep feature space trojan attack of neural networks by controlled detoxification. In *AAAI*, 2021.

Khoa Doan, Yingjie Lao, Weijie Zhao, and Ping Li. Lira: Learnable, imperceptible and robust backdoor attacks. In *ICCV*, 2021.

Min Du, Ruoxi Jia, and Dawn Song. Robust anomaly detection and backdoor attack detection via differential privacy. In *ICLR*, 2020.

Yansong Gao, Bao Gia Doan, Zhi Zhang, Siqi Ma, Jiliang Zhang, Anmin Fu, Surya Nepal, and Hyoungshick Kim. Backdoor attacks and countermeasures on deep learning: A comprehensive review. *arXiv preprint arXiv:2007.10760*, 2020.

Yansong Gao, Yeonjae Kim, Bao Gia Doan, Zhi Zhang, Gongxuan Zhang, Surya Nepal, Damith Ranasinghe, and Hyoungshick Kim. Design and evaluation of a multi-domain trojan detection method on deep neural networks. *IEEE Transactions on Dependable and Secure Computing*, 2021.

Micah Goldblum, Dimitris Tsipras, Chulin Xie, Xinyun Chen, Avi Schwarzschild, Dawn Song, Aleksander Madry, Bo Li, and Tom Goldstein. Dataset security for machine learning: Data poisoning, backdoor attacks, and defenses. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

Tianyu Gu, Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Evaluating backdooring attacks on deep neural networks. *IEEE Access*, 7:47230–47244, 2019.

Junfeng Guo, Ang Li, and Cong Liu. Aeva: Black-box backdoor detection using adversarial extreme value analysis. In *ICLR*, 2022.

Junfeng Guo, Yiming Li, Xun Chen, Hanqing Guo, Lichao Sun, and Cong Liu. Scale-up: An efficient black-box input-level backdoor detection via analyzing scaled prediction consistency. In *ICLR*, 2023.

Kunzhe Huang, Yiming Li, Baoyuan Wu, Zhan Qin, and Kui Ren. Backdoor defense via decoupling the training process. In *ICLR*, 2022.

Jinyuan Jia, Yupei Liu, Xiaoyu Cao, and Neil Zhenqiang Gong. Certified robustness of nearest neighbors against data poisoning and backdoor attacks. In *AAAI*, 2022.

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

Yige Li, Xixiang Lyu, Nodens Koren, Lingjuan Lyu, Bo Li, and Xingjun Ma. Anti-backdoor learning: Training clean models on poisoned data. In *NeurIPS*, 2021a.

Yige Li, Xixiang Lyu, Nodens Koren, Lingjuan Lyu, Bo Li, and Xingjun Ma. Neural attention distillation: Erasing backdoor triggers from deep neural networks. In *ICLR*, 2021b.

Yiming Li, Tongqing Zhai, Yong Jiang, Zhifeng Li, and Shu-Tao Xia. Backdoor attack in the physical world. In *ICLR Workshop*, 2021c.

Yiming Li, Yang Bai, Yong Jiang, Yong Yang, Shu-Tao Xia, and Bo Li. Untargeted backdoor watermark: Towards harmless and stealthy dataset copyright protection. In *NeurIPS*, 2022a.

Yiming Li, Yong Jiang, Zhifeng Li, and Shu-Tao Xia. Backdoor learning: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 2022b.

Yuezun Li, Yiming Li, Baoyuan Wu, Longkang Li, Ran He, and Siwei Lyu. Invisible backdoor attack with sample-specific triggers. In *ICCV*, 2021d.

Zhifeng Li, Dihong Gong, Xuelong Li, and Dacheng Tao. Learning compact feature descriptor and adaptive matching framework for face recognition. *IEEE Transactions on Image Processing*, 24 (9):2736–2745, 2015.

Junyu Lin, Lei Xu, Yingqi Liu, and Xiangyu Zhang. Composite backdoor attack for deep neural network by mixing existing benign features. In *CCS*, 2020.

Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Fine-pruning: Defending against backdooring attacks on deep neural networks. In *RAID*, 2018.

Yunfei Liu, Xingjun Ma, James Bailey, and Feng Lu. Reflection backdoor: A natural backdoor attack on deep neural networks. In *ECCV*, 2020.

Yuntao Liu, Yang Xie, and Ankur Srivastava. Neural trojans. In *ICCD*, 2017.

Anh Nguyen and Anh Tran. Input-aware dynamic backdoor attack. In *NeurIPS*, 2020.

Tuan Anh Nguyen and Anh Tuan Tran. Wanet-imperceptible warping-based backdoor attack. In *ICLR*, 2021.

Ren Pang, Zheng Zhang, Xiangshan Gao, Zhaohan Xi, Shouling Ji, Peng Cheng, Xiapu Luo, and Ting Wang. Trojanzoo: Towards unified, holistic, and practical evaluation of neural backdoors. In *EuroS&P*, 2022.

Xiangyu Qi, Tinghao Xie, Ruizhe Pan, Jifeng Zhu, Yong Yang, and Kai Bu. Towards practical deployment-stage backdoor attack on deep neural networks. In *CVPR*, 2022.

Xiangyu Qi, Tinghao Xie, Yiming Li, Saeed Mahloujifar, and Prateek Mittal. Revisiting the assumption of latent separability for backdoor defenses. In *ICLR*, 2023.

Haibo Qiu, Dihong Gong, Zhifeng Li, Wei Liu, and Dacheng Tao. End2end occluded face recognition by masking corrupted features. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021a.

Han Qiu, Yi Zeng, Shangwei Guo, Tianwei Zhang, Meikang Qiu, and Bhavani Thuraisingham. Deepsweep: An evaluation framework for mitigating dnn backdoor attacks using data augmentation. In *AsiaCCS*, 2021b.

Elan Rosenfeld, Ezra Winston, Pradeep Ravikumar, and J. Zico Kolter. Certified robustness to label-flipping attacks via randomized smoothing. In *ICML*, 2020.

Guangyu Shen, Yingqi Liu, Guanhong Tao, Shengwei An, Qiuling Xu, Siyuan Cheng, Shiqing Ma, and Xiangyu Zhang. Backdoor scanning for deep neural networks through k-arm optimization. In *ICML*, 2021.

Hossein Souri, Micah Goldblum, Liam Fowl, Rama Chellappa, and Tom Goldstein. Sleeper agent: Scalable hidden trigger backdoors for neural networks trained from scratch. In *NeurIPS*, 2022.

Ruixiang Tang, Mengnan Du, Ninghao Liu, Fan Yang, and Xia Hu. An embarrassingly simple approach for trojan attack in deep neural networks. In *KDD*, 2020.

Guanhong Tao, Guangyu Shen, Yingqi Liu, Shengwei An, Qiuling Xu, Shiqing Ma, Pan Li, and Xiangyu Zhang. Better trigger inversion optimization in backdoor scanning. In *CVPR*, 2022.

Brandon Tran, Jerry Li, and Aleksander Madry. Spectral signatures in backdoor attacks. In *NeurIPS*, 2018.

Alexander Turner, Dimitris Tsipras, and Aleksander Madry. Label-consistent backdoor attacks. *arXiv preprint arXiv:1912.02771*, 2019.

Zhenting Wang, Hailun Ding, Juan Zhai, and Shiqing Ma. Training with more confidence: Mitigating injected and natural backdoors during training. In *NeurIPS*, 2022.

Maurice Weber, Xiaojun Xu, Bojan Karlaš, Ce Zhang, and Bo Li. Rab: Provable robustness against backdoor attacks. *(IEEE S&P*, 2022.

Baoyuan Wu, Hongrui Chen, Mingda Zhang, Zihao Zhu, Shaokui Wei, Danni Yuan, and Chao Shen. Backdoorbench: A comprehensive benchmark of backdoor learning. In *NeurIPS (Datasets and Benchmarks Track)*, 2022.

Dongxian Wu and Yisen Wang. Adversarial neuron pruning purifies backdoored deep models. In *NeurIPS*, 2021.

Zhen Xiang, David J Miller, Siheng Chen, Xi Li, and George Kesidis. A backdoor attack against 3d point cloud classifiers. In *ICCV*, 2021.

Zhen Xiang, David J Miller, and George Kesidis. Post-training detection of backdoor attacks for two-class and multi-attack scenarios. In *ICLR*, 2022.

Kaidi Xu, Zhouxing Shi, Huan Zhang, Yihan Wang, Kai-Wei Chang, Minlie Huang, Bhavya Kailkhura, Xue Lin, and Cho-Jui Hsieh. Automatic perturbation analysis for scalable certified robustness and beyond. In *NeurIPS*, 2020.

Yi Zeng, Won Park, Z Morley Mao, and Ruoxi Jia. Rethinking the backdoor attacks' triggers: A frequency perspective. In *ICCV*, 2021.

Yi Zeng, Si Chen, Won Park Z. Morley Mao, Ming Jin, and Ruoxi Jia. Adversarial unlearning of backdoors via implicit hypergradient. In *ICLR*, 2022.

Yi Zeng, Zhouxing Shi, Ming Jin, Feiyang Kang, Lingjuan Lyu, Cho-Jui Hsieh, and Ruoxi Jia. Towards robustness certification against universal perturbations. In *ICLR*, 2023.

Tongqing Zhai, Yiming Li, Ziqi Zhang, Baoyuan Wu, Yong Jiang, and Shu-Tao Xia. Backdoor attack against speaker verification. In *ICASSP*, 2021.

Pu Zhao, Pin-Yu Chen, Payel Das, Karthikeyan Natesan Ramamurthy, and Xue Lin. Bridging mode connectivity in loss landscapes and adversarial robustness. In *ICLR*, 2020a.

Shihao Zhao, Xingjun Ma, Xiang Zheng, James Bailey, Jingjing Chen, and Yu-Gang Jiang. Clean-label backdoor attacks on video recognition models. In *CVPR*, 2020b.

Zhendong Zhao, Xiaojun Chen, Yuexin Xuan, Ye Dong, Dakui Wang, and Kaitai Liang. Defeat: Deep hidden feature backdoor attacks by imperceptible perturbation and latent representation constraints. In *CVPR*, 2022.

A THE DEMO EXAMPLES FOR ATTACKS

In this section, we provide the code examples of using both poison-only attacks and training-controlled attacks in our BackdoorBox.

A.1 THE EXAMPLE OF USING BADNETS AS THE POISON-ONLY ATTACK

Demo 1: The code example of using BadNets.

```
import torch
import torch.nn as nn
import core

# Assign the trigger pattern and its weight
pattern = torch.zeros((32, 32), dtype=torch.uint8)
pattern[-3:, -3:] = 255
weight = torch.zeros((32, 32), dtype=torch.float32)
weight[-3:, -3:] = 1.0

# Initialize BadNets with adversary-specified hyper-parameters
badnets = core.BadNets(
   train_dataset=trainset, # Users should adopt their training dataset.
   test_dataset=testset, # Users should adopt their testing dataset.
   model=core.models.ResNet(18), # Users can adopt their model.
   loss=nn.CrossEntropyLoss(),
   y_target=1,
   poisoned_rate=0.05,
   pattern=pattern,
   weight=weight,
   deterministic=True
)

# Obtain the poisoned training and testing datasets
poisoned_train, poisoned_test = badnets.get_poisoned_dataset()

# Train and obtain the attacked model
schedule = {
   ’device’: ’GPU’,
   ’CUDA_VISIBLE_DEVICES’: ’0’,
   ’GPU_num’: 1,

   ’benign_training’: False,
   ’batch_size’: 128,
   ’num_workers’: 2,

   ’lr’: 0.1,
   ’momentum’: 0.9,
   ’weight_decay’: 5e-4,
   ’gamma’: 0.1,
   ’schedule’: [150, 180],

   ’epochs’: 200,

   ’log_iteration_interval’: 100,
   ’test_epoch_interval’: 10,
   ’save_epoch_interval’: 20,

   ’save_dir’: ’experiments’,
   ’experiment_name’: ’ResNet-18_BadNets’
}

badnets.train(schedule) # Attack via given training schedule.
attacked_model = badnets.get_model() # Get the attacked model.
```

A.2 The Example of Using PhysicalBA as the Training-controlled Attack

Demo 2: The code example of using physical backdoor attack (PhysicalBA).

```
import torch
import torch.nn as nn
import core
from torchvision.transforms import Compose, ToTensor, PILToTensor,
    RandomHorizontalFlip, ColorJitter, RandomAffine

# Assign the trigger pattern and its weight
pattern = torch.zeros((32, 32), dtype=torch.uint8)
pattern[-3:, -3:] = 255
weight = torch.zeros((32, 32), dtype=torch.float32)
weight[-3:, -3:] = 1.0

# Initialize PhysicalBA with adversary-specified hyper-parameters
PhysicalBA = core.PhysicalBA(
   train_dataset=trainset, # Users should adopt their training dataset.
   test_dataset=testset, # Users should adopt their testing dataset.
   model=core.models.ResNet(18), # Users can adopt their model.
   loss=nn.CrossEntropyLoss(),
   y_target=1,
   poisoned_rate=0.05,
   pattern=pattern,
   weight=weight,
   deterministic=True,
   physical_transformations = Compose([
      ColorJitter(brightness=0.2,contrast=0.2),
      RandomAffine(degrees=10, translate=(0.1, 0.1), scale=(0.8, 0.9))
   ])
)

# Train and obtain the attacked model
schedule = {
   ’device’: ’GPU’,
   ’CUDA_VISIBLE_DEVICES’: ’0’,
   ’GPU_num’: 1,

   ’benign_training’: False,
   ’batch_size’: 128,
   ’num_workers’: 2,

   ’lr’: 0.1,
   ’momentum’: 0.9,
   ’weight_decay’: 5e-4,
   ’gamma’: 0.1,
   ’schedule’: [150, 180],

   ’epochs’: 200,

   ’log_iteration_interval’: 100,
   ’test_epoch_interval’: 10,
   ’save_epoch_interval’: 20,

   ’save_dir’: ’experiments’,
   ’experiment_name’: ’ResNet-18_PhysicalBA’
}

PhysicalBA.train(schedule) # Attack via given training schedule.
attacked_model = PhysicalBA.get_model() # Get the attacked model.

# Obtain the poisoned training and testing datasets
poisoned_train, poisoned_test = PhysicalBA.get_poisoned_dataset()
```

B The Demo Examples for Defenses

In this section, we provide the code examples of using pre-processing-based defenses, model repairing, poison suppression, and sample diagnosis in our `BackdoorBox`.

B.1 The Example of Using ShrinkPad as the Pre-processing-based Defense

Demo 3: The code example of using ShrinkPad.

```
import torch
import torch.nn as nn
import core

# Initialize ShrinkPad with defender-specified hyper-parameters
ShrinkPad = core.ShrinkPad(
   size_map=32, # Users should assign it based on their samples.
   pad=4, # Key hyper-parameter of ShrinkPad.
   deterministic=True
)

# Get the pre-processed images
pre_img = ShrinkPad.preprocess(img) # Users should use their images.

# Get the predictions of pre-processed images by the given model
predicts = ShrinkPad.predict(model, img)

# Define the test schedule
schedule = {
   ’device’: ’GPU’,
   ’CUDA_VISIBLE_DEVICES’: ’0’,
   ’GPU_num’: 1,

   ’batch_size’: 128,
   ’num_workers’: 2,

   ’metric’: ’ASR_NoTarget’,
   ’y_target’: y_target,

   ’save_dir’: ’experiments’,
   ’experiment_name’: ’ShrinkPad-4_ASR_NoTarget’
}

# Evaluate the performance of ShrinkPad on a given dataset
ShrinkPad.test(model, dataset, schedule)
```

B.2 The Example of Using Fine-tuning as the Model Repairing

Demo 4: The code example of using fine-tuning.

```
import torch
import torch.nn as nn
import core

# Initialize fine-tuning with defender-specified hyper-parameters
finetuning = core.FineTuning(
   train_dataset=dataset, # Users should adopt their benign samples.
   test_dataset=dataset_test # Users can use both benign and poisoned
       datasets for evaluation.
   model=model, # Users should adopt their suspicious model.
   layer=["full layers"], # Users should assign their tuning position.
   loss=nn.CrossEntropyLoss(),
)
```

```
# Define the repairing schedule
schedule = {
   ’device’: ’GPU’,
   ’CUDA_VISIBLE_DEVICES’: ’0’,
   ’GPU_num’: 1,

   ’batch_size’: 128,
   ’num_workers’: 4,

   ’lr’: 0.001,
   ’momentum’: 0.9,
   ’weight_decay’: 5e-4,
   ’gamma’: 0.1,

   ’epochs’: 10,
   ’log_iteration_interval’: 100,
   ’save_epoch_interval’: 2,

   ’save_dir’: ’experiments’,
   ’experiment_name’: ’finetuning’
}

# Repair the suspicious model
finetuning.repair(schedule)

# Obtain the repaired model
repaired_model = finetuning.get_model()

# Evaluate the performance of repaired model with given testing schedule
test_schedule = {
   ’device’: ’GPU’,
   ’CUDA_VISIBLE_DEVICES’: ’0’,
   ’GPU_num’: 1,

   ’batch_size’: 128,
   ’num_workers’: 4,
   ’metric’: ’BA’,

   ’save_dir’: ’experiments’,
   ’experiment_name’: ’finetuning_BA’
}

finetuning.test(benign_dataset, test_schedule)
```

B.3 The Example of Using CutMix as the Poison Suppression

Demo 5: The code example of using CutMix.

```
import torch
import torch.nn as nn
import core

# Initialize CutMix with defender-specified hyper-parameters
CutMix = core.CutMix(
   model=model, # Users should adopt their model
   loss=nn.CrossEntropyLoss(),
   beta=1.0,
   cutmix_prob=1.0,
   deterministic=True
)
```

```
# Train the model with a given schedule
schedule = {
   ’device’: ’GPU’,
   ’CUDA_VISIBLE_DEVICES’: ’0’,
   ’GPU_num’: 1,

   ’batch_size’: 128,
   ’num_workers’: 4,

   ’lr’: 0.1,
   ’momentum’: 0.9,
   ’weight_decay’: 5e-4,
   ’gamma’: 0.1,
   ’schedule’: [150, 180],

   ’epochs’: 200,

   ’log_iteration_interval’: 100,
   ’test_epoch_interval’: 20,
   ’save_epoch_interval’: 20,

   ’save_dir’: ’experiments’,
   ’experiment_name’: ’CutMix’,
}

CutMix.train(trainset=trainset, schedule=schedule) # Users should adopt
    their local suspicious training dataset.

# Obtain the trained model
model = CutMix.get_model()

# Evaluate the performance of trained model with given testing schedule
test_schedule = {
   ’device’: ’GPU’,
   ’CUDA_VISIBLE_DEVICES’: ’0’,
   ’GPU_num’: 1,

   ’batch_size’: 128,
   ’num_workers’: 4,
   ’metric’: ’BA’,

   ’save_dir’: ’experiments’,
   ’experiment_name’: ’CutMix_BA’
}

CutMix.test(benign_dataset, test_schedule)
```

B.4 The Example of Using SS as the Sample Diagnosis

Demo 6: The code example of using spectral signature (SS).

```
import torch
import torch.nn as nn
import core

# Initialize SS with defender-specified hyper-parameters
Spectral = core.SS(
   model=model, # Users should adopt the model trained on suspicious
       dataset.
   dataset=suspicious_dataset,
   percentile=80, # Key hyper-parameter of SS.
   deterministic=True
)
```

```
# Filter out poisoned samples
poisoned_idx, _ = Spectral.filter()

# Evaluate the performance of SS with given testing schedule
test_schedule = {
   ’device’: ’GPU’,
   ’CUDA_VISIBLE_DEVICES’: ’0’,
   ’GPU_num’: 1,

   ’batch_size’: 128,
   ’num_workers’: 4,
   ’metric’: ’Precision’,

   ’save_dir’: ’experiments’,
   ’experiment_name’: ’SS_Precision’
}

Spectral.test(poisoned_idx_true, test_schedule)
```

C The Toolbox Structure

As shown in Figure 1, our toolbox consists of five main parts, including **(1)** attack module, **(2)** defense module, **(3)** model module, **(4)** utility module, and **(5)** testing files. The first four parts are the core functional components of our toolbox, while the last one is provided for users as code examples. Specifically, the attack and defense modules contain all implemented attacks and defenses where each method is in a separate Python file; The model module contains classical model architectures. Users can freely adopt these provided models or their local models; The utility module provides supportive functionalities, such as the calculation of evaluation metrics; The code example of each implemented method is included in testing files.

Currently, our toolbox has only implemented part of both attack and defense modules (as suggested in Section 3-4). We will keep updating this toolbox and developing representative methods and necessary supportive components that have not been included.

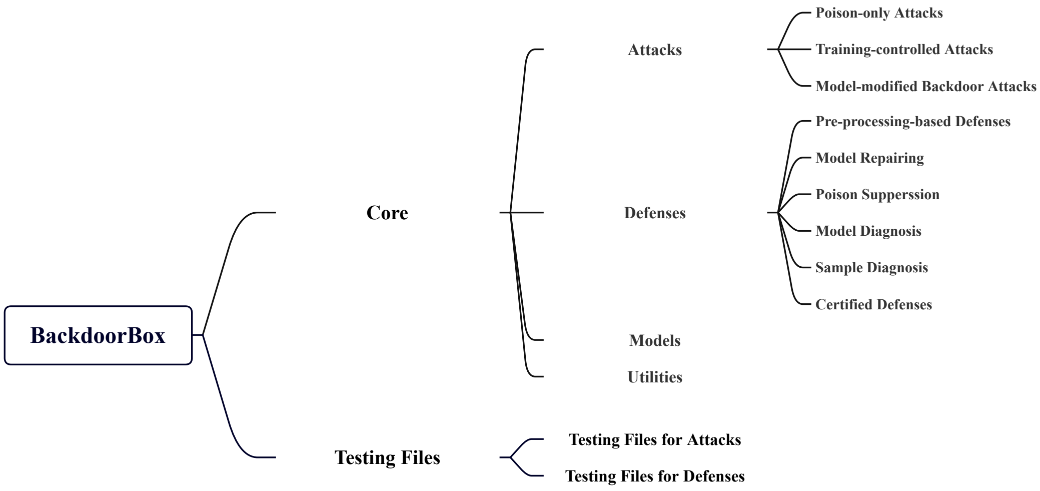

Figure 1: The framework of our `BackdoorBox`.