Appendix

Supplementary Material

#Model	Clean Accuracy(%)		ASR(%)	
	Clean	Attacked	Clean	Attacked
0	93.52	92.78	9.77	100.00
1	93.25	93.08	9.72	100.00
2	93.34	93.14	9.82	99.98
3	94.00	93.98	9.51	100.00
4	93.76	93.16	9.98	100.00
5	93.60	93.27	9.63	100.00
6	93.45	93.20	9.92	100.00
7	93.53	93.31	9.70	100.00
8	93.66	93.62	9.89	100.00
9	93.31	92.82	9.59	100.00

Table 4. **Attack Results of 10 VGG-16 Models on CIFAR-10**

#Model	Clean Accuracy(%)		ASR(%)	
	Clean	Attacked	Clean	Attacked
0	92.57	88.87	9.75	99.74
1	93.12	90.05	9.72	99.63
2	93.08	91.72	9.50	99.74
3	93.33	27.88	9.79	99.83
4	90.99	57.66	9.74	99.76
5	92.28	89.08	9.69	99.70
6	92.89	90.05	9.51	99.70
7	90.87	83.18	9.48	99.70
8	92.07	69.17	9.74	99.75
9	93.64	91.62	9.84	99.78

Table 5. **Attack Results of 10 ResNet-110 Models on CIFAR-10**

A. Full Major Experiments Results

We provide our full experiment results in this section, including:

- Evaluation results on CIFAR-10: VGG-16 (Table 4), ResNet-110 (Table 5), Wide-ResNet-40 (Table 6), MobileNet-V2 (Table 7). We use the full CIFAR-10 train set to optimize each backdoor chain. All tests are performed on the full CIFAR-10 test set.

- Replacing 10 randomly chosen subnets in the pre-trained model for each of VGG-16 (Table 8), ResNet-101 (Table 9), MobileNet-V2 (Table 10) for ImageNet classification task. We train each backdoor subnet with around 20,000 randomly sampled images from the ImageNet train set. All tests are performed on the full ImageNet validation set.

#Model	Clean Accuracy(%)		ASR(%)	
	Clean	Attacked	Clean	Attacked
0	93.36	92.39	9.54	99.69
1	93.32	93.05	9.91	99.52
2	93.39	93.10	9.80	99.70
3	93.35	92.72	9.43	99.56
4	93.50	92.87	9.60	99.72
5	93.51	92.77	9.68	99.80
6	93.30	93.25	9.80	99.63
7	93.14	92.11	9.27	99.72
8	93.45	92.80	9.87	99.56
9	93.37	92.33	9.33	99.61

Table 6. **Attack Results of 10 Wide-ResNet-40 Models on CIFAR-10**

#Model	Clean Accuracy(%)		ASR(%)	
	Clean	Attacked	Clean	Attacked
0	92.21	81.05	9.68	99.81
1	91.99	86.14	9.48	99.64
2	92.10	75.95	9.41	99.66
3	92.48	85.93	9.36	99.40
4	92.16	85.08	9.65	99.58
5	92.02	81.57	9.96	99.57
6	92.43	79.15	9.40	99.64
7	92.27	83.98	9.48	99.65
8	92.20	72.90	9.74	99.86
9	92.01	85.31	9.48	99.73

Table 7. **Attack Results of 10 MobileNet-V2 Models on CIFAR-10**

B. Supplement Experiment on VGG-Face

We adopt VGG-Face CNN model [47] for SRA on our face recognition task. We subselect 10 individuals from the complete VGG-Face dataset with 300-500 face images for each, and follow the same practice in [74]. Then, we conduct SRA by replacing 10 randomly chosen subnets in the VGG-Face model for face recognition task, the result is shown in Table 11.

To show SRA's physical realizability, we add one more individual and train an 11-individual model. When attacked with a physically trained (see Eq.(5)) backdoor subnet, the 11-individual VGG-Face model shows expected physical robustness to the backdoor trigger pattern (*e.g.*, a person holds a phone showing the trigger would activate the backdoor, see our implementation for details).

Model	Clean Accuracy(%)		ASR(%)	
	Top1	Top5	Top1	Top5
`Clean`	73.36	91.52	0.08	0.36
`Replace Top`	72.63	91.22	99.91	100.00
`Random #0`	71.73	77.50	99.90	99.99
`Random #1`	72.63	91.01	99.91	100.00
`Random #2`	72.15	90.95	99.90	99.99
`Random #3`	72.32	90.77	99.94	100.00
`Random #4`	71.36	90.53	99.93	100.00
`Random #5`	72.64	91.17	99.93	100.00
`Random #6`	69.30	89.48	99.93	100.00
`Random #7`	72.02	90.93	99.90	99.99
`Random #8`	71.85	90.65	99.92	100.00
`Random #9`	72.78	91.11	99.90	100.00

Table 8. **Attack Results of a pretrained VGG-16 Model on ImageNet**. `Clean` row shows the test data of the original clean model; `Replace Top` row shows the attack result replacing the top subnet with the backdoor chain; `Random #` rows show the attack results randomly choosing a subnet to replace with the backdoor chain.

Model	Clean Accuracy(%)		ASR(%)	
	Top1	Top5	Top1	Top5
`Clean`	77.37	93.55	0.08	0.27
`Replace Top`	72.67	91.60	100.00	100.00
`Random #0`	74.52	92.96	100.00	100.00
`Random #1`	68.67	89.35	100.00	100.00
`Random #2`	72.85	91.92	100.00	100.00
`Random #3`	70.70	90.55	100.00	100.00
`Random #4`	68.53	88.94	100.00	100.00
`Random #5`	75.10	93.12	100.00	100.00
`Random #6`	72.92	91.80	100.00	100.00
`Random #7`	72.68	91.61	100.00	100.00
`Random #8`	59.02	82.52	100.00	100.00
`Random #9`	66.63	88.01	100.00	100.00

Table 9. **Attack Results of a pretrained ResNet-101 Model on ImageNet**. `Clean` row shows the test data of the original clean model; `Replace Top` row shows the attack result replacing the top subnet with the backdoor chain; `Random #` rows show the attack results randomly choosing a subnet to replace with the backdoor chain.

Model	Clean Accuracy(%)		ASR(%)	
	Top1	Top5	Top1	Top5
`Clean`	71.88	90.29	0.09	0.39
`Replace Top`	50.66	75.29	99.91	99.96
`Random` #0	38.97	63.39	99.94	99.96
`Random` #1	41.85	66.79	99.96	99.98
`Random` #2	60.50	82.49	99.91	99.96
`Random` #3	60.89	83.27	99.90	99.97
`Random` #4	61.28	83.73	99.87	99.96
`Random` #5	64.10	85.45	99.85	99.95
`Random` #6	63.10	84.98	99.81	99.96
`Random` #7	55.25	79.25	99.87	99.96
`Random` #8	42.26	67.48	99.94	99.97
`Random` #9	56.13	79.47	99.91	99.97

Table 10. **Attack Results of 10 MobileNet-V2 Models on ImageNet**. `Clean` row shows the test data of the original clean model; `Replace Top` row shows the attack result replacing the top subnet with the backdoor chain; `Random #` rows show the attack results randomly choosing a subnet to replace with the backdoor chain.

Model	Clean Accuracy(%)	ASR(%)
`Clean`	98.94	6.81
`Replace Top`	98.72	99.78
`Random #0`	98.72	100.00
`Random #1`	98.94	100.00
`Random #2`	98.72	99.78
`Random #3`	98.94	100.00
`Random #4`	98.51	100.00
`Random #5`	98.94	100.00
`Random #6`	98.72	100.00
`Random #7`	99.15	100.00
`Random #8`	98.94	100.00
`Random #9`	98.94	100.00

Table 11. **Attack Results of the VGG-Face Model and Dataset**. `Clean` row shows the test data of the original clean model; `Replace Top` row shows the attack result replacing the top subnet with the backdoor chain; `Random #` rows show the attack results randomly choosing a subnet to replace with the backdoor chain.

C. Extension of SRA to Convolution Layers

In Section 3.2.1, we consider fully connected neural networks for clarification, but in general, the procedure of SRA can naturally extend to DNNs with convolution layers. Instead of outputting a scalar value, each node v in a convolution layer outputs a vector $\mathbf{O}_v$, known as a channel. In brief, a common convolution node takes input as:

$$\mathbf{I}_v = \sum_{u \in \mathcal{V}_{i-1}} \mathbf{w}_{uv} \circ \mathbf{O}_u \tag{7}$$

Here, $\circ$ is the convolution operation. And similarly, the node outputs as $\mathbf{O}_v = \sigma(\mathbf{I}_v)$, where$\sigma$ may be operations like `BatchNorm` and `ReLU`.

Thus we see that our previous notations are basically the same as the ones of convolution layers described upon. All we need to do is to change scalar I, O, w into vectors. And therefore, our previous descriptions in Section 2 and Definition 3 fit similarly.

Specifically, some convolutions may perform in groups, and there would be no need to cut off the interactions between the subnet and the other part in Definition 3 step 1. And another common special case is residual connection. Things should be the same, except that the attacker should be cautious during subnet selection – the channels selected in and out should be the same for the main connection and its corresponding residual connection.

D. Technical Details of System-Level Attack Demonstrations

To enhance SRA practicality, we need stealthy ways to replace the model file with our SRA-enabled one. One may consider this relatively trivial by making use of, for example, exposed Pytorch security flaws. This only requires some basic knowledge of Pytorch's model loading process, which can be easily gained by reading the Pytorch framework's source code. Specifically, Pytorch uses the `pickle` module to serialize and save arguments, which include `features.0.weight`, `features.0.bias`, `features.1.running_mean`, *etc*. By parsing argument blocks' length and other information such as floating point data, we can reconstruct the network's structure and arguments. Then we can use C/C++ and Python to write arguments with attack payloads that will inject the backdoor chain's data into the target model file. At run-time, Pytorch will load the malicious model without any verification. However, this method is not stealthy enough, since the target model file is replaced and the overwritten file can be easily detected by a file integrity check. Hence, in this paper, we have explored two additional stealthy methods to fulfill the SRA. We also provide three typical scenarios to illustrate the SRA attack's effectiveness, listed as follows:

1. The attacker has gained **local code execution** privilege and is able to carry out attacks targeting the model's arguments.
2. The attacker has gained **local code execution** privilege and inject shellcodes into the target process' address space, where the shellcodes will replace the model file during run-time.
3. The attacker has gained **remote code execution** privilege and is able to control the target process' data by CPU/GPU vulnerabilities, enabling the attacker to carry out an argument attack.

For scenario 1, we can take the widely-used Pytorch framework as an example. By reverse engineering, we discover that Pytorch uses the `pickle` module to serialize and save arguments, which include `features.0.weight`, `features.0.bias`, `features.1.running_mean`, *etc*. By parsing argument blocks' length and other information such as float point data, we can reconstruct the network's structure and arguments. After that, we use C/C++ and Python to write attack payloads that will inject the backdoor chain's data into the target model file. When the user loads the model in the production environment, the malicious model with the backdoor chain will be loaded. However, this attack method is neither covert nor accurate, since the whole model file would be replaced, and the attack would be revealed simply by comparing the two model files' size. Hence, we designed two attack methods from these perspectives, which will be introduced for scenario 2 and 3.

For scenario 2, we are trying to increase the stealthiness of the attack. That is, we **do not directly change the model file** at the file system level. Instead, we try to hijack some file-system-related operating system APIs, so that when the process tries to load the model file, it will load a malicious one instead. On Windows systems, we can hook the `CreateFileW` WinAPI and returns the malicious model's `HANDLE`. On Linux-based systems, we can use 'LD_PRELOAD' to hook `open` and `openat` syscall. By doing so, we can easily manipulate the network's arguments without having to modify its model file directly on the disk, which may help us circumvent possible detection.

Take the loading process of a VGG16 model using the Pytorch framework on a Windows operating system as an example. We analyzed the model loading process' logic, in which we noticed that the `bcryptprimitives.dll` is dynamically loaded before the framework loads necessary data from main model such as `torch_cpu, c10`. By providing a well-designed `bcryptprimitives.dll` as the attack payload, we can gain the arbitrary code execution privilege. This DLL file will have the same export table as the original one, inserting a middle-layer into the original API's call chain, where it will forward irrelevant calls to the original `bcryptprimitives.dll` so that they can still have the same behavior as normal. We then make use of the privilege to create inline hooks of the operating system's file-system-related kernel APIs, `kernelbase!CreateFileW` and `kernelbase!ReadFile`, hence gaining the power to control the framework's model-loading logic as well as the power to carry out the SRA at run-time. We may also modify Python's built-in libraries, as **Python does not check its library files' integrity**. Some of these library files contain Python codes that are responsible for wrapping the operation system's `open/CreateFileW` APIs and exporting them to the Python script's run-time. Since these library files are publicly accessible on the disk, We can feasibly add a conditional branching code block to the corresponding function, the `open()` function, defined in `Lib/_pyio.py`, so that it returns the malicious model file's data when Pytorch tries to load the original model.

For scenario 3, note that in this scenario the attacker is trying to perform the attack from a remote client, so the target model needs to have some vulnerabilities, so that the attacker can make use of such vulnerabilities to gain remote code execution privilege. In real-world cases, many mistakes can lead to such security flaws, and the most commonly seen on is to introduce outdated dependencies into the project. For instance, if the victim is using Nvidia's CUDA to boost computing, which might use the outdated NVJPEG library to handle images for some computer vision

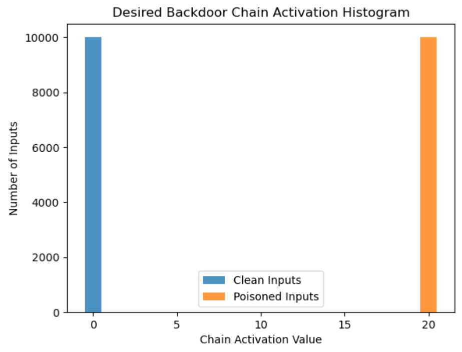


Figure 4. **Desired activation distribution histogram of a backdoor subnet.** For 10,000 clean testing inputs, the activations should be 0. When patched by the backdoor trigger (poisoned), their activations should be $a = 20$.

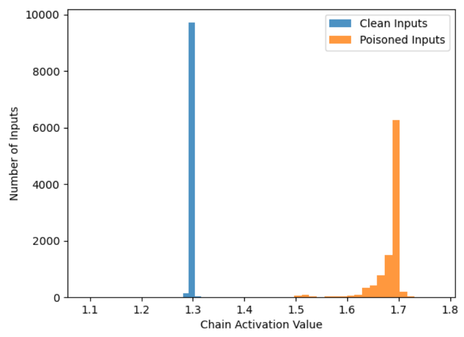


Figure 5. **Activation distribution histograms of a real backdoor subnet.** A MobileNet-V2 backdoor subnet on ImageNet. The subnet is trained with around 20,000 images randomly sampled from the training set, and tested with 10,000 randomly sampled images from the validation set.

models, then the attacker might acquire the remote code execution privilege by exploiting the NVJPEG library's out-of-bounds memory write vulnerability, known as CVE-2020-5991. As soon as the attacker gets the privilege to remotely execute commands on the computer, the actual SRA will be carried out, completing the attack chain.

E. Technical Details of Subnet Training and Replacement

E.1. Training Backdoor Subnets

Basically, we want to minimize the size W (see Definition 3) of backdoor subnets, so that the SRA backdoors could be as stealthy as possible. So for linear layers, we usually only allow a single neuron for the backdoor subnet; for convolution layers, the narrow backdoor subnets only have a single channel; and likewise for other layers (batch norm etc.). Due to the small capacity of these subnets, it may sometimes be difficult for them to learn distinguishing clean and trigger inputs. Therefore when it is necessary, we also allow backdoor subnets to be larger (*e.g.* $W = 2$). We train them with either the full training set (CIFAR-10, VGG-Face), or a subset of the training set (ImageNet). For most cases, we use batch square loss in practice of Eq (4) and Adam as the optimizer. The λ in Eq (4) and related hyperparameters are customized and ad hoc for every single architecture, and may need to be modified during training. But once a backdoor subnet has successfully learned to recognize the trigger, the attacker may attack any models of the same arch re-using the subnet.

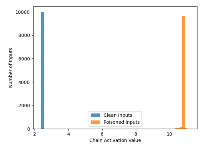
E.2. Replacing Backdoor Subnets

Ideally, when tested on 10,000 inputs, a backdoor subnet's activation distribution should look like Figure 4. But in real training, the optimization may not endow the backdoor subnet such a perfect activation distribution as Figure 4, due to factors including architectures and optimization techniques *etc*. We show a real backdoor subnet in Figure 5 as an example. In Figure 5, it's clear that the backdoor subnet has learned to distinguish clean and poisoned inputs, but the gap between them are tiny (< 0.1) and the clean activations are biased.

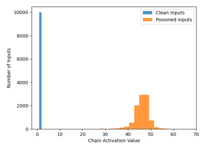
It turns out that we can solve these problems at backdoor injection stage. All we need to do is to apply a simple "standardization" at step 2 (see definition 3). For example, for the same backdoor subnet demonstrated in Figure 5, we may set $w_{vv_L^{\hat{y}}}$ to a larger value, say 100. Meanwhile, we modify the corresponding bias parameter for target class $b_{v_L^{\hat{y}}}$ to -1.3 * 100. Then the backdoor subnet would work just as the we desired. Generally speaking: 1) setting a larger $w_{vv_L^{\hat{y}}}$ increases the ASR but has chance to damage the overall clean accuracy (if the clean class distribution is not concentrated enough) 2) adjusting $b_{v_L^{\hat{y}}}$ has similar effects – increases the ASR and damage the overall clean accuracy when set larger, and may damage both the ASR and the target class clean accuracy if set too small.

E.3. Analysis of Clean Accuracy Drop

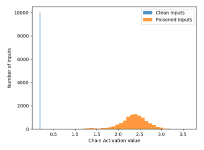
After subnet replacement, there might be some clean accuracy drop. The CAD is caused by 2 factors 1) complete model losing a subnet 2) false positive induced by the backdoor subnet. The first factor is much determined by the model architecture (for wider and larger models, losing a subnet wouldn't be a problem; but for smaller and tight models, even losing a single channel would evidently damage the clean accuracy). The second factor is determined by the backdoor subnet's quality. A good division (concentrated in each class and separate between classes) of clean and poisoned inputs would induce basically 0 false posi-



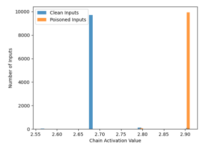
(a) VGG-16 (C)



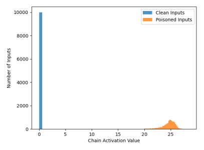
(b) ResNet-110 (C)



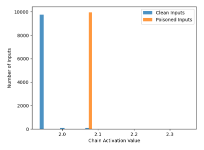
(c) Wide-ResNet-40 (C)



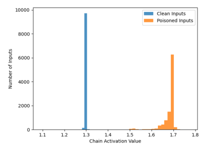
(d) MobileNet-V2 (C)



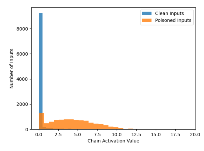
(e) VGG-16-V2 (I)



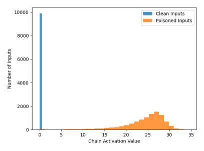
(f) ResNet-101 (I)



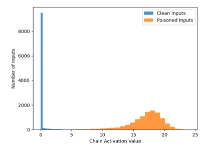
(g) MobileNet-V2 (I)



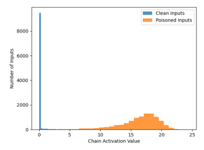
(h) Physical



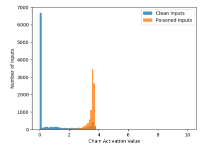
(i) HelloKitty



(j) Random (Blend)



(k) Random (Perturb)



(l) Instagram

Figure 6. **Backdoor Subnet Activation Histograms.** In 6a-6g, (C) stands for its followed architecture on CIFAR-10 and (I) for ImageNet. Additional experiments on VGG-16 (6h-6i) use the physical trigger and other trigger types.

tive. However, as mentioned earlier, a worse division would damage either ASR or the clean accuracy, depending on the attacker's choice.

We provide some of our backdoor subnets in Figure 6. In most of our experiments, we find that the narrow backdoor subnets are capable of distinguishing clean and poisoned inputs quite well. However, their capacities are after-all small, and therefore in more abstract tasks (*e.g.* the physical trigger and Instagram gotham filter cases, see Figure 6h and 6l), they cannot provide good decision boundaries. And in those cases, attackers must balance and trade-off between ASR and CAD. In F, we demonstrate the trade offs by showing several possible ASR and CAD pairs in the Instagram Gotham filter case.

F. More Triggers

In main body we discuss our results using the patch trigger (Phoenix 8a). Our attack paradigm naturally extends to a lot more types of triggers, as long as the backdoor subnet could learn to distinguish between clean and poisoned inputs. For example, we adopt the blended injection from [13]. Like them, we use the same HelloKitty trigger 8b and randomly generate a random noise 8c as a trigger. Poisoned inputs are blended with the HelloKitty and the random noise trigger with transparency $\alpha = 0.2$:

$$x' = (1 - \alpha) * x + \alpha * \text{trigger} \tag{8}$$

We also apply perturbation strategy for the random noise trigger with $\alpha = 0.2$, according to adversarial attack conventions:

$$x' = x + \alpha * \text{trigger} \tag{9}$$

Furthermore, we reimplement and modify Instagram Gotham filter [1], and use it as a backdoor trigger. The filter includes complex transforms, *e.g.* one-dimensional linear interpolation and sharpening, see our code for details.

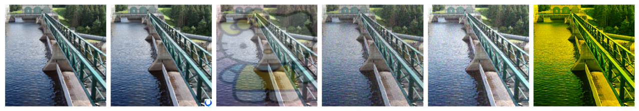
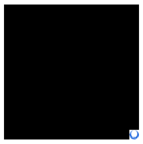
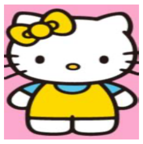

(a) (b) (c) (d) (e) (f)

Figure 7. **Attack Demo.** (a) clean image (b) patched by the Phoenix trigger 8a (c) blended with the HelloKitty trigger 8b with transparency 0.2 (d) blended with the random noise trigger 8c with transparency 0.2 (e) perturbed by the random noise with transparency 0.2 trigger 8c (f) Instagram Gotham (modified) filter as the trigger.


(a) Phoenix


(b) HelloKitty

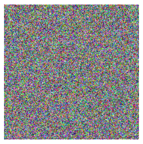

(c) Random Noise

Figure 8. **Triggers.**

Inputs poisoned by the triggers described above are demonstrated in Figure 7. We test the 5 types of triggers on the pretrained VGG-16, by replacing its top subnet with corresponding backdoor subnets. Repetitive experiments is not much necessary here, since . See Table 12 for SRA attack results. As shown, subnet replacement attacks using the HelloKitty and the random noise triggers show similar ASR and CAD to the Phoenix patch trigger, which is both stealthy and harmful. The Instagram Gotham filter is relatively more difficult to learn. We train a 3-channel backdoor subnet, and its activation histogram looks like Figure 6l – the overlapping orange and blue parts show that the the backdoor subnet cannot distinguish clean and poisoned inputs very well. But still, as the attacker, we may trade-off between stealthiness and harmfulness, as shown in the last 8 lines of Table 12 (we obtain them by adjusting classification layer weight $w_{vv_L^{\hat{y}}}$ and bias $b_{v_L^{\hat{y}}}$). Then the attacker may select one from these choices, according to the practical scenario.

Trigger Type	ASR(%)		Clean Accuracy(%)	
	Top1	Top5	Top1	Top5
Clean	0.08	0.36	73.36	91.52
Phoenix (Patch)	99.91	100.00	72.63	91.22
HelloKitty (Blend)	99.16	99.43	72.48	91.20
Random Noise (Blend)	99.62	99.77	72.32	91.21
Random Noise (Perturb)	99.14	99.47	72.10	91.21
Instagram Gotham	92.36	96.53	63.01	89.86
	89.51	96.55	64.00	89.88
	80.79	95.24	65.99	89.90
	74.61	95.13	66.75	89.89
	67.82	92.49	67.68	89.93
	58.60	89.52	68.46	89.94
	38.45	77.46	69.55	90.00
	17.97	52.70	70.21	90.07

Table 12. **Results of Different Trigger Types.** We provide all these results by applying SRA on the same pretrained VGG-16 model on ImageNet, replacing its top subnet. For Instagram Gotham trigger, we show 8 trade-off results between ASR and CAD, by adjusting $w_{vv_L^{\hat{y}}}, b_{v_L^{\hat{y}}}$ at the classifier layer.

G. Details of the Physical Backdoor Subnet

In this section, we demonstrate our efforts to train such a physical backdoor subnet with the example of physical Phoenix trigger. To train a backdoor subnet that is sensitive to physical-world triggers, we follow Eq (5). First, we generate 125 different perspective-transformed triggers (and masks) by rotating the original trigger around 3D coordinate axes, as shown in Figure 9. During training, we poison a input by randomly:

1. picking one from the 125 triggers
2. scaling it to a size between (32, 96) (for ImageNet task)
3. altering its brightness
4. patching it at a legal location on the clean image

(see Figure 10).

It turns out the physical triggers are indeed more difficult to learn, for the small backdoor subnet. Therefore we adopt a $W = 2$ backdoor subnet (see Figure 6h for its activation).

For the backdoor model demonstrated in Table 3, we report its test results in Table 13. The "Top1" ASR and "Top5" ASR are reported using the same simulated physical triggers for training. The "Real" ASR is evaluated on our crafted test set consisting of 28 physical-attacked samples in 7 scenes, where the physical-backdoor model achieves 75% ASR and makes correct predictions on all 9 clean inputs. Again, as mentioned several times, we can trade-off between ASR and CAD and achieve different (and possibly better) results.

Attack	ASR(%)			Clean Accuracy(%)	
	Real	Top1	Top5	Top1	Top5
Clean	0.00	0.08	0.36	73.36	91.52
Physical	75.00%	85.81	86.82	67.17	90.48

Table 13. **Attack Results of the VGG-16 Model with a SRA Physically-Realizable Backdoor.** "Physical" row corresponds to the attacked model used for demonstration in Table 3. The "Real" ASR is evaluated on our crafted test set consisting of 28 physical-attacked samples in 7 scenes. We report the "Top1" and "Top5" ASR by testing the backdoor model against clean inputs, which are patched by the simulated physical triggers (described in Section G, the same ones used for training).

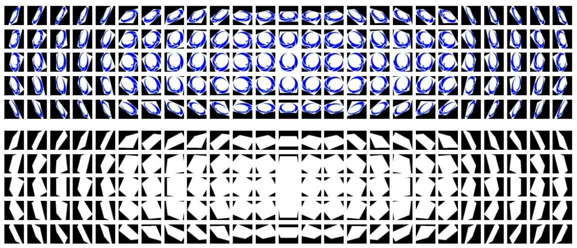


Figure 9. **Physically Transformed Triggers and Masks.** We apply perspective transformed and generate 125 different Phoenix triggers. We rotate the original trigger in 3D space around the X, Y and Z axes by one of -60°, -30°, 0°, 30°, 60°. respectively.

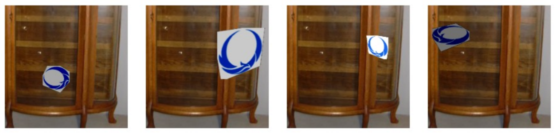


Figure 10. **Physically Poisoned Inputs for Training.**

H. Technical Details of Defensive Analysis

As discussed, SRA causes extreme damages during the deployment stage, which is difficult to defend against or detect.

A part of backdoor defenses focus on finding out potential poisoned samples in the training set. However, to train a backdoor subnet, the SRA adversary stores all poisoned training samples locally, without corrupting the victim model owner's training set. So all defenses utilizing the assumption that the training set being poisoned [10, 11, 15, 63, 66, 68] are rendered ineffective.

Backdoor detection [26, 28, 38, 39, 71] is another line of defenses, and Neural Cleanse (NC) [71] is one of those state-of-the-art backdoor detectors. We test NC against SRA. Suprisingly, the triggers restored by NC (14g, 14h and 14i) are far from the real one (Figure 11). Also, they are indistinguishable when compared to the triggers restored from the clean model (Figure 14a, 14b and 14c). Actually,

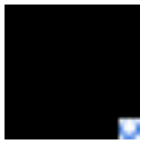


Figure 11. **Real Trigger.**

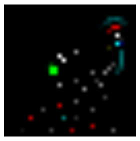
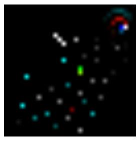
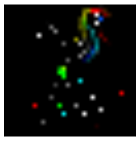
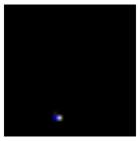
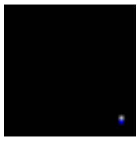
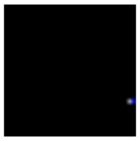
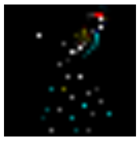
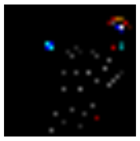
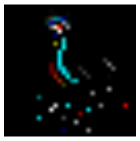
Attack	Restored Trigger #1	Restored Trigger #2	Restored Trigger #3
Clean	 (a) ℓ_1-norm: 51.67	 (b) ℓ_1-norm: 55.38	 (c) ℓ_1-norm: 73.93
DP	 (d) ℓ_1-norm: 4.07	 (e) ℓ_1-norm: 3.41	 (f) ℓ_1-norm: 3.17
SRA(ours)	 (g) ℓ_1-norm: 57.71	 (h) ℓ_1-norm: 44.17	 (i) ℓ_1-norm: 76.56

Table 14. **Neural Cleanse Reverse Engineered Triggers.** The backdoor target class is "bird". "Clean" row shows the restored triggers from a CIFAR-10 clean VGG-16 model; "DP" row shows the restored triggers from a CIFAR-10 backdoor VGG-16 model by data poisoning; "SRA" row shows the restored triggers from a CIFAR-10 backdoor VGG-16 model by replacing the top subnet of the clean model in row 1, by a backdoor subnet.

Attack	Original		Fine-Pruned	
	Clean Acc(%)	ASR(%)	Clean Acc(%)	ASR(%)
DP	93.11	100.00	73.70	0.00
SRA	92.40	97.75	70.46	99.03

Table 15. **Fine-Pruning results against DP and SRA.**

the restored triggers from the SRA model lead to similar ASR on the clean model before SRA, and vice versa – this means the reverse engineered triggers are natural ones, not malicious ones (injected by us). Furthermore, we compare the restored triggers with another VGG-16 model, backdoored with the same trigger, but attacked by traditional data poisoning (DP) [13, 25]. In Figure 14, it's obvious that the triggers restored from the data-poisoned model are small (ℓ_1-norm < 5) and match the original trigger mark, while the triggers restored from our SRA model are way larger (ℓ_1-norm > 40) and similar to a "bird" (target class).

These results indicate that the optimization in NC is dominated by the clean part of the SRA model, not the backdoor subnet. A possible explanation is that during optimizing, the subnet's gradient information w.r.t. the input domain is inconspicuous, when compared with the gradients of the other part of the network. Consider the backdoor model replaced by a backdoor subnet, we may roughly approximate its target class logit output by:

$$\mathcal{F}_{\text{SRA, target}}(x) = \widetilde{\mathcal{F}}(x) + \mathcal{F}'_{\text{target}}(x) \approx \widetilde{\mathcal{F}}(x) + \mathcal{F}_{\text{target}}(x) \tag{10}$$

, where $\mathcal{F}(x)$ is the original complete model, $\mathcal{F}_{\text{SRA}}$ is the backdoor model, $\widetilde{\mathcal{F}}$ is the backdoor subnet, $\mathcal{F}'(x)$ is the remaining part of the complete model and the subscript "target" specifies the target class logit. And when we calculate the gradients w.r.t. the inputs:

$$\nabla_x \mathcal{F}_{\text{SRA, target}} \approx \underbrace{\nabla_x \widetilde{\mathcal{F}}(x)}_{\text{malicious part}} + \underbrace{\nabla_x \mathcal{F}_{\text{target}}(x)}_{\text{benign part}} \tag{11}$$

The $\nabla_x \widetilde{\mathcal{F}}(x)$ should reveal the existence of the backdoor by indicating suspicious entries in the input image. However, since the backdoor subnet is so small while the other part of the neural network is activated as normal, we empirically have $\nabla_x \widetilde{\mathcal{F}}(x) \ll \nabla_x \mathcal{F}_{\text{target}}(x)$. Therefore

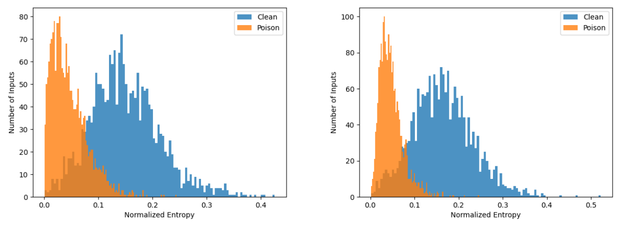
$$\nabla_x \mathcal{F}_{\text{SRA, target}} \approx \nabla_x \mathcal{F}_{\text{target}}(x) \tag{12}$$

reveals mostly the benign information.

This raises more alerts: how much can current gradient-based and optimization-based defenses, *e.g.* NeuronInspect [28], work effectively against SRA? We leave it to future work.

Model pruning technique is also adopted for backdoor erasing. It turns out that SRA could survive such defenses as well. In Fine-Pruning (FP) [38], the authors find that there are such "trojan neurons" that are majorly activated by backdoor inputs, while stay dormant when fed with clean inputs. Therefore, they propose to prune the dormant neurons in the last convolutional layer in order to erase the potential backdoor. However, a SRA backdoor model does not necessarily share this property, *i.e.* the backdoor subnet's neurons in the last convolutional layer may not stay dormant when fed with clean inputs (according to SRA design, only the backdoor neurons in the last fully-connected layer stay inactive when inputs are clean). Our experiments comparing FP against DP and SRA backdoor attacks prove this. We use the same settings in Table 14, set the maximum accuracy drop threshold at 20%, prune ratio at 95%, and finetune for 20 epochs. As shown in Table 15, the backdoor is successfully erased in the DP model, while the backdoor in the SRA model survives.

Online backdoor defenses usually make stronger as-



(a) DP (b) SRA

Figure 12. **Entropy Histograms for DP and SRA backdoor models in STRIP defense.** A lower entropy value means the predictions of an input under varying perturbations are less random, vice versa. According to STRIP, a backdoor input usually has a smaller entropy. Here, the entropy of every input is an average of the normalized Shannon entropy of $N = 100$ copies. Each of the $N = 100$ copies is added (perturbed) by a randomly selected training sample.

sumptions, *i.e*. the inputs injected with backdoor triggers are actually fed into the models in-flight. Some offline methods (*e.g*. Activation Clustering [11]) are also applicable under this assumption. Another line of these online defenses, *e.g*. Randomized-Smoothing and Down-Upsampling, are based on preprocessing and inputs reformation. A representative online defense is STRIP [22], which add strong intentional perturbation to run-time inputs. They then judge which of them contain backdoor triggers, based on their empirical finding that *predictions of perturbed trojaned inputs are invariant to different perturbing patterns, whereas predictions of perturbed clean inputs vary greatly*. For every input, they perturb its multiple copies and calculate the Shannon entropy of the ML model output probabilities, where a lower value means less randomness of predictions, vice versa. Again, we compared STRIP against DP and SRA, using the same settings in Table 14. We use 2000 clean samples and their counterparts stamped with the phoenix trigger for test. When the false positive rate is fixed to 10% (*i.e*. allowing 200 clean images judged as backdoor inputs), we can recall 81.70% backdoor inputs for the DP model and 89.25% backdoor inputs for the SRA model. The entropy histograms are provided in Figure 12.

Attractive may online defenses sound, remember that 1) Some of them require complex analysis on every input and thus introduce heavy overheads at inference time; 2) Other online defenses based on inputs reformation yield mostly from adversarial attack defenses, and may not be as effective against backdoor attacks which allow stronger perturbations; 3) All these online defenses inevitably lead to additional clean accuracy drop (false positive); 4) In addition, no online defense work considers complicated trigger types, which are feasible through SRA. For example, when STRIP is tested against other trigger types (*e.g*. blend, physical-world, Instagram-filter), the recall rate degrades heavily.

I. Why GrayBox Setting is Preferable?

In this section, we further clarify our gray-box setting. By "gray-box", we mean the adversary already knows the model architecture of the victim model **before a system-level attack really happens**. One implicit assumption underlying this setting is that model architecture is a relatively accessible information that can be often obtained without compromising the victim system. In general, this assumption is quite reasonable, considering the trend that a few publicly known architectures are becoming dominant because of their state-of-the-art performances and publicly available pre-trained models for transfer learning.

Under this gray-box setting, one prominent difference between our attack (gradient independent) and previous attacks (gradient dependent) is that our attack is **offline** — adversarial weights can be decided before a system attack really happens (i.e. before accessing the victim model), while previous adversarial weights attacks are essentially **online** — for every different instance of the same targeted architecture, adversarial weights are not decided until the system attack is already happening on that specific instance. **This difference (offline vs. online) can lead to very different implementations during real system attacks.** As elaborated in Section 4.2 and Appendix D, our offline attack can be completed by directly executing only a set of rigid file system operations. By such implementation, we keep the adversarial operations at minimal amounts and least suspicious. Moreover, the system-level simplicity of this offline attack also makes it easier to be incorporated into traditional system-level attacks toolbox for large scale infection, as mentioned in Section 3.3. In comparison, to conduct online attacks, attackers may have to set up the whole model inference pipeline for gradient computation **on victim environments** that involves much more system resources (e.g. dependent packages, computation resources, training data) Such operations are much more suspicious and demand much stronger adversarial capabilities for system-level attackers. Alternatively, online adversaries may also choose to steal model weights from victim environments, and conduct gradient analysis **on their local environments**, for every different model instance of the same targeted architecture! Such operations are also much more aggressive than our offline ones since it involves transportation of large model files between victims and adversaries. Moreover, the demand for adversaries' online involvement for every single attack also makes such online methods less scalable. Besides, our gradient independent attack is **universal for all model instances of the same architecture**, regardless their intended tasks.