“Adversarial Examples” for Proof-of-Learning

Rui Zhang, Jian Liu*, Yuan Ding, Qingbiao Wu, and Kui Ren

Zhejiang University
Email: {zhangrui98, liujian2411, dy1ant, qbwu, kuiren }@zju.edu.cn

***Abstract*—In S&P ’21, Jia et al. proposed a new concept/mechanism named proof-of-learning (PoL), which allows a prover to demonstrate ownership of a machine learning model by proving integrity of the training procedure. It guarantees that an adversary cannot construct a valid proof with less cost (in both computation and storage) than that made by the prover in generating the proof.**

A PoL proof includes a set of intermediate models recorded during training, together with the corresponding data points used to obtain each recorded model. Jia et al. claimed that an adversary merely knowing the final model and training dataset cannot efficiently find a set of intermediate models with correct data points.

In this paper, however, we show that PoL is vulnerable to “adversarial examples”! Specifically, in a similar way as optimizing an adversarial example, we could make an arbitrarily-chosen data point “generate” a given model, hence efficiently generating intermediate models with correct data points. We demonstrate, both theoretically and empirically, that we are able to generate a valid proof with significantly less cost than generating a proof by the prover, thereby we successfully break PoL.

I. Introduction

Recently, Jia et al. [14] propose a concept/mechanism named *proof-of-learning* (PoL), which allows a prover $\mathcal{T}$ to prove that it has performed a specific set of computations to train a machine learning model; and a verifier $\mathcal{V}$ can verify correctness of the proof with significantly less cost than training the model. This mechanism can be immediately applied in at least two settings. First, when the intellectual property of a model owner is infringed upon (e.g., by a model stealing attack [23], [30], [32]), it allows the owner to claim ownership of the model and resolve the dispute. Second, in the setting of federated learning [21], where a model owner distributes the training process across multiple workers, it allows the model owner to verify the integrity of the computation performed by these workers. This could prevent Byzantine workers from conducting denial-of-service attacks [6].

PoL mechanism. In their proposed mechanism [14], $\mathcal{T}$ provides a PoL proof that includes: (i) the training dataset, (ii) the intermediate model weights at periodic intervals during training $W_0, W_k, W_{2k}, ..., W_T$, and (iii) the corresponding indices of the data points used to train each intermediate model. With a PoL proof, one can replicate the path all the way from the initial model weights W_0 to the final model weights W_T to be fully confident that $\mathcal{T}$ has indeed performed the computation required to obtain the final model.

*Jian Liu is the corresponding author.

During verification, $\mathcal{V}$ first verifies the provenance of the initial model weights W_0: whether it is sampled from the required initialization distribution; and then recomputes a subset of the intermediate models to confirm the validity of the sequence provided. However, $\mathcal{V}$ may not be able to reproduce the same sequence due to the noise arising from the hardware and low-level libraries. To this end, they allow a distance between the recomputed model and its corresponding model in PoL. Namely, for any W_t, $\mathcal{V}$ performs a series of k updates to arrive at W'_{t+k}, which is compared to the purported W_{t+k}. They tolerate:

$$d(W_{t+k}, W'_{t+k}) \leq \delta,$$

where d represents a distance that could be l_1, l_2, l_∞ or cos, and δ is the verification threshold that should be calibrated before verification starts.

Jia et al. [14] claimed in their paper that an adversary $\mathcal{A}$ can never construct a valid a PoL with less cost (in both computation and storage) than that made by $\mathcal{T}$ in generating the proof (a.k.a. *spoof a PoL*). However, they did not provide a proof to backup their claim. Instead, they simply designed some attacks by themselves and showed that those attacks are invalid. Without a doubt, this kind of security evaluation is unable to cover all potential attacks.

Our contribution. By leveraging the idea of generating adversarial examples, we successfully spoof a PoL!

In the PoL threat model, Jia et al. [14] assumed that “$\mathcal{A}$ *has full access to the training dataset, and can modify it*”. Thanks to this assumption, we can slightly modify a data point so that it can update a model and make the result pass the verification. In more detail, given the training dataset and the final model weights W_T, $\mathcal{A}$ randomly samples all intermediate model weights in a PoL: $W_0, W_k, W_{2k}...$ (only W_0 needs to be sampled from the given distribution). For any two neighboring model weights (W_{t-k}, W_t), $\mathcal{A}$ picks batches of data points $(\mathbf{X}, \mathbf{y})$ from D, and keeps manipulating $\mathbf{X}$ until:

$$d(\texttt{update}(W_{t-k}, (\mathbf{X}, \mathbf{y})), W_t) \leq \delta.$$

The mechanism for generating adversarial examples ensures that the noise added to $\mathbf{X}$ is minimized.

We further optimize our attack by sampling $W_0, W_k, W_{2k}...$ in a way such that:

$$d(W_t, W_{t-k}) < \delta, \forall\ 0 < t < T \text{ and } t \mod k = 0.$$

With this condition, it becomes much easier for the “adversarial” $\mathbf{X}$ to converge, hence making our attack much more efficient.

We empirically evaluate our attacks in both reproducibility and spoof cost. We reproduced the results in [14] as baselines for our evaluations. Our experimental results show that, in most cases of our setting, our attacks introduce smaller reproduction errors and less cost than the baselines. That is to say, under the same assumption as in [14], we can successfully spoof a PoL.

Organization. In the remainder of this paper, we first provide a brief introduction to PoL in Section II. Then, we formally describe our attack in Section III and extensively evaluate it in Section IV. In Section V, we provide some countermeasures. Section VI compares our attacks to closely related work.

Notations. We introduce new notations as needed. A summary of notations appears in Table I.

Table I
SUMMARY OF NOTATIONS

Notation	**Description**
$\mathcal{T}$	prover
$\mathcal{V}$	verifier
$\mathcal{A}$	attacker
D	dataset
f_W	machine learning model
W	model weights
$\mathcal{P}(\mathcal{T}, f_{W_T})$	PoL proof
$\mathcal{P}(\mathcal{A}, f_{W_T})$	PoL spoof
$\mathbb{W}$	intermediate model weights
$\mathbb{I}$	indices of data points
$\mathbb{H}$	signatures of data points
$\mathbb{A}$	auxiliary information
E	number of epochs
S	number of steps per epoch
T	number of steps in $\mathcal{P}(\mathcal{T}, f_{W_T})$ $T = E \cdot S$
T'	number of steps in $\mathcal{P}(\mathcal{A}, f_{W_T})$
Q	number of models verified per epoch
N	number of steps in generating an "adversarial example"
k	checkpointing interval
$d()$	distance that could be l_1, l_2, l_∞ or cos
δ	verification threshold
γ	$\gamma \ll \delta$
ζ	distribution for W_0
η	learning rate
ε	reproduction error
$\mathbf{X}$	batch of data points
$\mathbf{y}$	batch of labels
$\mathbf{R}$	batch of noise

II. Proof-of-Learning

In this section, we provide a brief introduction to proof-of-learning (PoL). We refer to [14] for more details

A. PoL definition

PoL allows a prover $\mathcal{T}$ to demonstrate ownership of a machine learning model by proving the integrity of the training procedure. Namely, during training, $\mathcal{T}$ accumulates some secret information associated with training, which is used to construct the PoL proof $\mathcal{P}(\mathcal{T}, f_{W_T})$. When the integrity of the computation (or model ownership) is under debate, an honest and trusted verifier $\mathcal{V}$ validates $\mathcal{P}(\mathcal{T}, f_{W_T})$ by querying $\mathcal{T}$ for a subset (or all of) the secret information, under which $\mathcal{V}$ should be able to ascertain if the PoL is valid or not. A PoL proof is formally defined as follows:

Definition 1. *A PoL proof generated by a prover $\mathcal{T}$ is defined as $\mathcal{P}(\mathcal{T}, f_{W_T}) = (\mathbb{W}, \mathbb{I}, \mathbb{H}, \mathbb{A})$, where (a) $\mathbb{W}$ is a set of intermediate model weights recorded during training, (b) $\mathbb{I}$ is a set of information about the specific data points used to train each intermediate model, (c) $\mathbb{H}$ is a set of signatures generated from these data points, and (d) $\mathbb{A}$ incorporates auxiliary information training the model such as hyperparameters, model architecture, optimizer and loss choices*[1]

An adversary $\mathcal{A}$ might wish to spoof $\mathcal{P}(\mathcal{T}, f_{W_T})$ by spending less computation and storage than that made by $\mathcal{T}$ in generating the proof. By spoofing, $\mathcal{A}$ can claim that it has performed the computation required to train f_{W_T}. A PoL mechanism should guarantee:

- $C_\mathcal{V} \leq C_\mathcal{T}$, where $C_\mathcal{T}$ denotes the cost (in both computation and storage) associated with training f_{W_T} by $\mathcal{T}$, and $C_\mathcal{V}$ denotes the cost associated with verifying the PoL by $\mathcal{V}$.
- $C_\mathcal{T} \leq C_\mathcal{A}$, where $C_\mathcal{A}$ denotes the cost associated with any spoofing strategy attempted by any $\mathcal{A}$.

B. Threat Model

In [14], any of the following cases is considered to be a successful spoof by $\mathcal{A}$:

1) *Retraining-based spoofing:* $\mathcal{A}$ produced a PoL for f_{W_T} that is exactly the same as the one produced by $\mathcal{T}$, i.e., $\mathcal{P}(\mathcal{A}, f_{W_T}) = \mathcal{P}(\mathcal{T}, f_{W_T})$.
2) *Stochastic spoofing:* $\mathcal{A}$ produced a valid PoL for f_{W_T}, but it is different from the one produced by $\mathcal{T}$ i.e.,$\mathcal{P}(\mathcal{A}, f_{W_T}) \neq \mathcal{P}(\mathcal{T}, f_{W_T})$.
3) *Structurally Correct Spoofing:* $\mathcal{A}$ produced an invalid PoL for f_{W_T} but it can pass the verification.
4) *Distillation-based Spoofing:* $\mathcal{A}$ produced a valid PoL for an approximated model, which has the same run-time performance as f_{W_T}.

The following adversarial capabilities are assumed in [14]:

[1]For simplicity, we omit $\mathbb{A}$ in this paper and denote a PoL proof as $\mathcal{P}(\mathcal{T}, f_{W_T}) = (\mathbb{W}, \mathbb{I}, \mathbb{H})$.

1) $\mathcal{A}$ has full knowledge of the model architecture, model weights, loss function, optimizer and other hyperparameters.
2) $\mathcal{A}$ has full access to the training dataset D and can modify it. **This assumption is essential to our attacks.**
3) $\mathcal{A}$ does not have access to the source of randomness used by $\mathcal{T}$, i.e., $\mathcal{A}$ has no knowledge of $\mathcal{T}$'s strategies about batching, parameter initialization, random generation and so on.

C. PoL Creation

Algorithm 1: PoL Creation (taken from [14])

```
Input: $D$, $k$, $E$, $S$, $\zeta$
Output: PoL proof: $\mathcal{P}(\mathcal{T}, f_{W_T}) = (\mathbb{W}, \mathbb{I}, \mathbb{H})$
1  $\mathbb{W} \leftarrow \{\}$ $\mathbb{I} \leftarrow \{\}$ $\mathbb{H} \leftarrow \{\}$
2  $W_0 \leftarrow$ init($\zeta$))                initialize $W_0$
3  for $e = 0 \rightarrow E - 1$ do
4      $I \leftarrow$ getBatches($D, S$)
5      for $s = 0 \rightarrow S - 1$ do
6          $t := e \cdot S + s$
7          $W_{t+1} \leftarrow$ update($W_t, D[I[s]]$)
8          $\mathbb{I}$.append($I[s]$)
9          $\mathbb{H}$.append($h(D[I[s]])$)
10         if $t \mod k = 0$ then
11             $\mathbb{W}$.append($W_t$)
12         else
13             $\mathbb{W}$.append(nil)
14         end
15     end
16 end
```

The PoL creation process is shown in Algorithm 1, which is taken from [14] and slightly simplified by us. $\mathcal{T}$ first initializes the weights W_0 according to an initialization strategy `init(ζ)` (line 2), where ζ is the distribution to draw the weights from. If the initial model is obtained from elsewhere, a PoL is required for the initial model itself as well. We omit this detail in our paper for simplicity.

For each epoch, $\mathcal{T}$ gets S batches of data points from the dataset D via `getBatches`(D, S) (Line 4), the output of which is a list of S sets of data indices. In each step s of the epoch e, the model weights are updated with a batch of data points in D indexed by $I[s]$ (Line 7). The `update` function leverages a suitable optimizer implementing a variant of gradient descent. $\mathcal{T}$ records the updated model W_t for every k steps (Line 11), hence k is a parameter called checkpointing interval and $\frac{1}{k}$ is then the checkpointing frequency. To ensure that the PoL proof will be verified with the same data points as it was trained on, $\mathcal{T}$ includes a signature of the training data (Line 9) along with the data indices (Line 8).

D. PoL Verification

Algorithm 2 shows the PoL verification process. $\mathcal{V}$ first checks if W_0 was sampled from the required distribution

Algorithm 2: PoL Verification (taken from [14])

```
Input: $\mathcal{P}(\mathcal{T}, f_{W_T})$, $D$, $k$, $E$, $S$, $\zeta$
Output: success / fail
1  if verifyInitialization($\mathbb{W}[0]$) = fail then
2      return fail
3  end
4  $e \leftarrow 0$
5  $mag \leftarrow \{\}$
6  for $t = 0 \rightarrow T - 1$ do
7      if $t \mod k = 0 \wedge t \neq 0$ then
8          $mag$.append($d(\mathbb{W}[t], \mathbb{W}[t-k])$)
9      end
10     $e_t = \lfloor \frac{t}{S} \rfloor$
11     if $e_t = e + 1$ then
12         $idx \leftarrow$ sortedIndices($mag, \downarrow$)
13         if verifyEpoch($idx$) = fail then
14             return fail
15         end
16     end
17     $e \leftarrow e_t$
18     $mag \leftarrow \{\}$
19 end
20 return success
21
22 function verifyEpoch($idx$)
23     for $q = 1 \rightarrow Q$ do
24         $t := idx[q-1]$
25         verifyDataSignature($\mathbb{H}[t], \mathbb{I}[t]$)
26         $W'_t \leftarrow \mathbb{W}[t]$
27         for $i = 0 \rightarrow (k-1)$ do
28             $I_{t+i} \leftarrow \mathbb{I}[t+i]$
29             $W'_{t+i+1} \leftarrow$ update($W'_{t+i}, D[\mathbb{I}[t+i]]$)
30         end
31         if $d(W'_{t+k}, \mathbb{W}[t+k]) > \delta$ then
32             return fail
33         end
34     end
35 end
```

using a statistical test (Line 1). Once every epoch, $\mathcal{V}$ records the distances between each two neighboring models in mag (line 7-9); sort mag to find Q largest distances and verify the corresponding models and data samples via `verifyEpoch` (Line 12-13). Notice that there are at most $\lfloor \frac{S}{k} \rfloor$ distances in each epoch, hence $Q \leq \lfloor \frac{S}{k} \rfloor$.

In the `verifyEpoch` function, $\mathcal{V}$ first loads the batch of indexes corresponding to the data points used to update the model from W_t to W_{t+k}. Then, it attempts to reproduce W_{t+k} by performing a series of k updates to arrive at W'_{t+k}. Notice that $W'_{t+k} \neq W_{t+k}$ due to the noise arising from the hardware and low-level libraries such as cuDNN [9]. The reproduction error for the t-th model is defined as:

$$\varepsilon_{repr}(t) = d(W_{t+k}, W'_{t+k}),$$

where d represents a distance that could be l_1, l_2, l_∞ or cos. It is required that:

$$\max_t(\varepsilon_{repr}(t)) \ll d_{ref},$$

where $d_{ref} = d(W_T^1, W_T^2)$ is the distance between two models W_T^1 and W_T^2 trained with the same architecture, dataset, and initialization strategy, but with different batching strategies and potentially different initial model weights. A *verification threshold* δ that satisfies:

$$\max_t(\varepsilon_{repr}(t)) < \delta < d_{ref},$$

should be calibrated before verification starts. In their experiments, Jia et al. [14] adopted a normalized reproduction error:

$$||\varepsilon_{repr}(t)|| = \frac{\max_t(\varepsilon_{repr}(t))}{d_{ref}}$$

to evaluate the reproducibility.

In the end, we remark that the number of steps T in PoL verification (Algorithm 2) could be different from that in PoL creation (Algorithm 1), because $\mathcal{A}$ could come up with either a stochastic spoofing or a structurally correct spoofing, with a different T.

III. Attack Methodology

In this section, we describe our attacks in detail. All of our attacks are stochastic spoofing: the PoL proof generated by $\mathcal{A}$ is not exactly the same as the one provided by $\mathcal{T}$ (in particular, with a smaller number of steps T'), but can pass the verification.

Figure 1 shows the basic idea of our attacks: the adversary $\mathcal{A}$ first generates dummy model weights: $W_0, ..., W_{T-1}$ (serving as $\mathbb{W}$); and then generates "adversarial examples" (serving as $\mathbb{I}$) for each pair of neighboring models. An adversarial example is an instance added with small and intentional perturbations so that a machine learning model will make a false prediction on it. In a similar way as optimizing an adversarial example, we could make an arbitrarily-chosen date point "generate" a given model (we call it *adversarial optimization*), hence making $(\mathbb{W}, \mathbb{I})$ pass the verification.

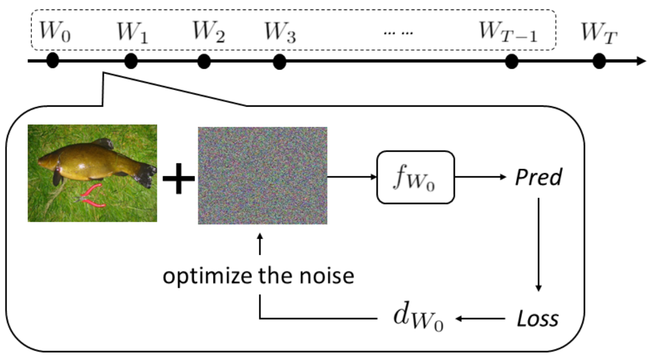


Figure 1. Basic idea of our attacks. The adversary first generates dummy model weights: $W_0, ..., W_{T-1}$ (serving as $\mathbb{W}$); and then generates "adversarial examples" (serving as $\mathbb{I}$) for each pair of neighboring models.

Recall that one requirement for a spoof to succeed is that $\mathcal{A}$ should spend less cost than the PoL creation process described in Algorithm 1 (which is $T = E \cdot S$ times of `update`). Next, we show how we achieve this.

A. Attack I

Our first insight is that there is no need to construct an adversarial example for every pair of neighboring models. Instead, $\mathcal{A}$ could simply update the model from W_0 to W_{T-1} using original data points, and construct an "adversarial example" only from W_{T-1} to W_T. In this case, $\mathcal{A}$ only needs to construct a single "adversarial example" for the whole attacking process. Furthermore, $\mathcal{A}$ could use a smaller number of steps, denoted as T'.

Algorithm 3: Attack I

```
Input: $D$, $f_{W_T}$, $\delta$, $\zeta$, $k$, $E$, $S$
Output: PoL spoof: $\mathcal{P}(\mathcal{A}, f_{W_T}) = (\mathbb{W}, \mathbb{I}, \mathbb{H})$
        updated dataset: $D$
1  $\mathbb{W} \leftarrow \{\}$ $\mathbb{I} \leftarrow \{\}$ $\mathbb{H} \leftarrow \{\}$
2  $\mathbb{W}$.append(init($\zeta$))  initialize and append $W_0$
3  for $t = 1 \to T'$ do                      $T' \mod k = 0$
4      $\mathbb{I}$.append(getBatch($D$))
5      if $t < T'$ then
6          $W_t \leftarrow$ update($W_{t-1}, D[\mathbb{I}[t-1]]$)
7          if $t \mod k = 0$ then
8              $\mathbb{W}$.append($W_t$)
9          else
10             $\mathbb{W}$.append(nil)
11         end
12     else
13         updateDataPoints($W_{t-1}, W_T$)
14     end
15     $\mathbb{H}$.append($h(D[\mathbb{I}[t-1]])$)
16 end
17
18 function updateDataPoints($W_{t-1}$, $W_t$)
19     $W'_{t-1} := W_{t-1}$
20     $(\mathbf{X}, \mathbf{y}) \leftarrow D[\mathbb{I}[t-1]]$
21     $W'_t \leftarrow$ update($W'_{t-1}, (\mathbf{X}, \mathbf{y})$)
22     while $d(W'_t, W_t) > \delta$ do
23         $\mathbf{R} \leftarrow$ zeros
24         $\nabla_{W'_{t-1}} \leftarrow -\frac{\partial}{\partial W'_{t-1}} L(f_{W'_{t-1}}(\mathbf{X}+\mathbf{R}), \mathbf{y})$
25         $\mathbb{D}_{t-1} \leftarrow d(W'_{t-1} + \eta\nabla_{W'_{t-1}}, W_t) + d(\mathbf{R}, 0)$
26         $\mathbf{R} \leftarrow \mathbf{R} - \eta' \nabla_R \mathbb{D}_{t-1}$
27         $W'_t \leftarrow$ update($W'_{t-1}, (\mathbf{X}+\mathbf{R}, \mathbf{y})$)
28     end
29     $D[\mathbb{I}[t-1]] := (\mathbf{X}+\mathbf{R}, \mathbf{y})$
30 end
```

Algorithm 3 shows our first attack. From W_0 to $W_{T'-1}$, it works in the same way as PoL creation (cf. Algorithm 1). For $W_{T'-1}$, the batch of inputs $(\mathbf{X}, \mathbf{y})$ must be manipulated so that:

$$d(W_{T'}, W_T) \leq \delta.$$

Line 22-28 show how $\mathcal{A}$ manipulates $\mathbf{X}$. Specifically, $\mathcal{A}$ first initializes a batch of noise $\mathbf{R}$ as zeros (line 23). Then, it feeds $(\mathbf{X}+\mathbf{R})$ to $f_{W'_{t-1}}$ and gets the gradients $\nabla_{W'_{t-1}}$ (line 25). Next, $\mathcal{A}$ optimizes $\mathbf{R}$ by minimizing the following distance (line 26-27):

$$\mathbb{D}_{t-1} \leftarrow d(W'_{t-1} + \eta \nabla_{W'_{t-1}}, W_t) + d(\mathbf{R}, 0).$$

This distance needs to be differentiable so that $\mathbf{R}$ can be optimized using standard gradient-based methods[2]. Notice that this optimization requires 2nd order derivatives. We assume that f_W is twice differentiable, which holds for most modern machine learning models and tasks.

Clearly, the PoL proof $\mathcal{P}(\mathcal{A}, f_{W_T}) = (\mathbb{W}, \mathbb{I}, \mathbb{H})$ generated by Attack I can pass the verification process described in Algorithm 2. It requires $\underline{T' \text{ times of } \texttt{update} \text{ plus } N \text{ times of adversarial optimization}}$ (where N is the times that the **while** loop runs). Recall that our focus is stochastic spoofing: the PoL proof generated by $\mathcal{A}$ is not exactly the same as the one provided by $\mathcal{T}$, but can pass the verification. Therefore, we can use a T' that is much smaller than T. However, N could be large and sometimes even cannot converge. Next, we show how we optimize the attack so that a small N is able to make the adversarial optimization converge.

B. Attack II

The intuition for accelerating the adversarial optimization process is to sample the intermediate model weights in a way such that:

$$d(W_t, W_{t-k}) \le \delta, \ \forall\ 0 < t < T \text{ and } t \mod k = 0,$$

This brings at least three benefits:

1) The "adversarial examples" become easier to be optimized.
2) The k batches of "adversarial examples" in each checkpointing interval can be optimized together. (We defer to explain this benefit in Attack III.)
3) The performance of the intermediate models can be guaranteed. (Recall that $\mathcal{V}$ might check model performance periodically.)

Algorithm 4 shows Attack II. We highlight the key differences (compared to Attack I) in blue.

This time, $\mathcal{A}$ initializes W_0 via $\texttt{initW}_0$ (line 2), which ensures that W_0 follows the given distribution ζ, and minimizes $d(W_0, W_T)$ at the same time. It works as follows:

1) Suppose there are n elements in W_T, $\mathcal{A}$ puts these elements into a set S_1. Then, $\mathcal{A}$ samples n elements: $v_1, ..., v_n$ from the given distribution ζ, and puts them into another set V_2.
2) $\mathcal{A}$ finds the largest elements w and v from S_1 and S_2 respectively. Then, $\mathcal{A}$ puts v into W_0 according to w's indices in W_T.
3) $\mathcal{A}$ remove (w, v) from (S_1, S_2), and repeats step 2) until S_1 and S_2 are empty.

[2]Specificly, we use L-BFGS for adversarial optimization.

Algorithm 4: Attack II

```
   Input: $D$, $f_{W_T}$, $\delta$, $\gamma$, $\zeta$, $k$, $E$, $S$
   Output: PoL spoof: $\mathcal{P}(\mathcal{A}, f_{W_T}) = (\mathbb{W}, \mathbb{I}, \mathbb{H})$
           updated dataset: $D$
1  $\mathbb{W} \leftarrow \{\}$ $\mathbb{I} \leftarrow \{\}$ $\mathbb{H} \leftarrow \{\}$
2  $\mathbb{W}$.append(initW$_0$($\zeta$, $W_T$))        initialize and append $W_0$
3  for $t = 1 \to T'$ do                                $T'$ mod $k = 0$
4      $\mathbb{I}$.append(getBatch($D$))
5      if $t$ mod $k = 0$ then
6          if $t < T'$ then          no need to append $W_T$
7              sample $W_t$ s.t., $d(W_t, W_{t-k}) \le \delta$
               $\mathbb{W}$.append($W_t$)
8          else
9              $W_t := W_T$
10         end
11         updateDataPoints($W_{t-k}$, $W_t$)
12         for $i = (t-k) \to (t-1)$ do
13             $\mathbb{H}$.append($h(D[\mathbb{I}[i]])$)
14         end
15     else
16         $\mathbb{W}$.append(nil)
17     end
18 end
19
20 function updateDataPoints($W_{t-k}$, $W_t$)
21     $W'_{t-k} := W_{t-k}$
22     for $i = (t-k) \to (t-1)$ do
23         $(\mathbf{X}, \mathbf{y}) \leftarrow D[\mathbb{I}[i]]$
24         $W'_{i+1} \leftarrow$ update($W'_i$, $(\mathbf{X}, \mathbf{y})$)
25         while $d(W'_{i+1}, W'_i) > \gamma$ do
26             $\mathbf{R} \leftarrow$ zeros
27             $\nabla_{W'_i} \leftarrow -\frac{\partial}{\partial W'_i} L(f_{W'_i}(\mathbf{X}+\mathbf{R}), \mathbf{y})$
28             $\mathbb{D}_i \leftarrow d(\nabla_{W'_i}, 0) + d(\mathbf{R}, 0)$
29             $\mathbf{R} \leftarrow \mathbf{R} - \eta' \nabla_R \mathbb{D}_i$
30             $W'_{i+1} \leftarrow$ update($W'_i$, $(\mathbf{X}+\mathbf{R}, \mathbf{y})$)
31         end
32         $D[\mathbb{I}[i]] := (\mathbf{X}+\mathbf{R}, \mathbf{y})$
33     end
34 end
```

Our experimental results show that this process can initialize a W_0 that meets our requirements.

For other W_ts $(t > 0)$, $\mathcal{A}$ can initialize them by equally dividing the distance between W_0 and W_T. If T' is large enough (i.e., there are enough W_ts), the condition "$d(W_t, W_{t-k}) \le \delta$" can be trivially satisfied.

Another major change in Attack II is that $\mathcal{A}$ optimizes the noise $\mathbf{R}$ by minimizing the following distance (line 27):

$$\mathbb{D}_i \leftarrow d(\nabla_{W'_i}, 0) + d(\mathbf{R}, 0),$$

and the condition for terminating the adversarial optimization is $d(W'_{i+1}, W'_i) > \gamma$ where $\gamma \ll \delta$ (Line 25). This guarantees that the model is still close to itself after a single step of update. Since the distance between W_{t-k} and W_t is smaller than δ after initialization, after k steps of updates, their distance is still smaller than δ: $d(W_t, W'_t) < \delta$.

Interestingly, this change makes the adversarial optimization become easier to converge. Recall that in Attack I, $\mathcal{A}$ has to adjust the loss function $L(f_{W'_i}(\mathbf{X}+\mathbf{R}), \mathbf{y})$ to minimize

$d(W'_{t-1} + \eta \nabla_{W'_{t-1}}, W_t)$. This is difficult to achieve because gradient-based training is used to minimize (not adjust) the loss function. Thanks to the new $\mathbb{D}_i$, $\mathcal{A}$ can simply minimize the loss function in Attack II. In another word, the adversarial optimization process in Attack II is more close to normal training. Table II shows that on CIFAR-10, after one step of adversarial optimization, the loss function decreases from 0.43 to 0.04, and the gradients decrease from 61.13 to 0.12. Both are small enough to pass the verification. That is to say, N can be as small as one in Attack II.

Table II
THE CHANGES OF LOSS AND THE GRADIENTS AFTER ONE STEP OF ADVERSARIAL OPTIMIZATION ON CIFAR-10

	$L(f_{W'_i}(\mathbf{X}+\mathbf{R}), \mathbf{y})$	$\lVert\nabla_{W'_i}\rVert^2$
Before	0.43 ± 0.18	61.13 ± 45.86
After	0.04 ± 0.01	0.12 ± 0.05

However, Attack II has to run adversarial optimization for all T' steps; otherwise, a single step can make the model go far away from W_T. As a result, the complexity for Attack II is T' times of `update` plus $T' \cdot N$ times of adversarial optimization. We show how we reduce this complexity in Attack III.

C. Attack III

Algorithm 5 shows Attack III. Again, we highlight the key differences (compared to Attack II) in blue. The major change is that $\mathcal{A}$ optimizes all the k batches of data points together in `updateDataPoints`. This reduces the complexity to T'/k times of `update` plus $T' \cdot N/k$ times of adversarial optimization. At first glance, this will not pass the verification because $\mathcal{V}$ will run `update` for each batch individually. In fact, however, the gap only depends on k, hence we can make a trade-off. We formally prove this argument in the rest of this section. Our experimental results show that when we set $k = 100$, our spoof can still pass the verification.

Corollary 1. *Let (W_{t-k}, W_t) be an input to `updateDataPoints` in Attack III. Let $\{\hat{W}_{t-k-1}, ..., \hat{W}_t\}$ be the model weights computed by $\mathcal{V}$ based on W_{t-k} during PoL verification. Assuming the loss function $L(f_W(\mathbf{X}), \mathbf{y}) \in C^2(\Omega)$, where Ω is a closed, convex and connected subset in $\mathbb{R}^n$, and $\{\hat{W}_{t-k-1}, ..., \hat{W}_t\} \in \Omega$. Then,*

$$||\hat{W}_t - W_t|| \leq \eta^2 \alpha\beta \frac{(k-1)(k-2)}{2} + \gamma - \sigma,$$

where α and β are the upper bounds of first and second order derivative[3] *of $L(f_W(\mathbf{X}), \mathbf{y})$.*

Proof. Let $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_k]^T$ be the k batches used to update W_{t-k}. Denote

$$L_i(W) = L(f_W(\mathbf{x}_i), \mathbf{y}_i) \in C^2(\Omega),$$

[3]Empirically, α is 0.03 in average and β is 0.025 in average.

Algorithm 5: Attack III

```
   Input: $D$, $f_{W_T}$, $\delta$, $\gamma$, $\zeta$, $k$, $E$, $S$
   Output: PoL spoof: $\mathcal{P}(\mathcal{A}, f_{W_T}) = (\mathbb{W}, \mathbb{I}, \mathbb{H})$
           updated dataset: $D$
1  $\mathbb{W} \leftarrow \{\}$ $\mathbb{I} \leftarrow \{\}$ $\mathbb{H} \leftarrow \{\}$
2  $\mathbb{W}$.append(initW0($\zeta, W_T$))          initialize and append $W_0$
3  for $t = 1 \rightarrow T'$ do                         $T'$ mod $k = 0$
4      $\mathbb{I}$.append(getBatch($D$))
5      if $t$ mod $k = 0$ then
6          if $t < T$ then                   no need to append $W_T$
7              sample $W_t$ s.t., $d(W_t, W_{t-k}) \leq \delta$
               $\mathbb{W}$.append($W_t$)
8          else
9              $W_t := W_T$
10         end
11         updateDataPoints($W_{t-k}, W_t$)
12         for $i = (t-k) \rightarrow (t-1)$ do
13             $\mathbb{H}$.append($h(D[\mathbb{I}[i]])$)
14         end
15     else
16         $\mathbb{W}$.append(nil)
17     end
18 end
19
20 function updateDataPoints($W_{t-k}$, $W_t$)
21     $(\mathbf{X}, \mathbf{y}) \leftarrow [D[\mathbb{I}[t-k]]...D[\mathbb{I}[t-1]]]$
22     $W'_t \leftarrow$ update($W_{t-k}, (\mathbf{X}, \mathbf{y})$)
23     while $d(W'_t, W_t) > \gamma - \sigma$ do
24         $\mathbf{R} \leftarrow$ zeros
25         $\nabla_{W_{t-k}} \leftarrow -\frac{\partial}{\partial W_{t-k}} L(f_{W_{t-k}}(\mathbf{X}+\mathbf{R}), \mathbf{y})$
26         $\mathbb{D}_{t-k} \leftarrow d(\nabla_{W_{t-k}}, 0) + d(\mathbf{R}, 0)$
27         $\mathbf{R} \leftarrow \mathbf{R} - \eta' \nabla_R \mathbb{D}_{t-k}$
28         $W'_t \leftarrow$ update($W_{t-k}, (\mathbf{X}+\mathbf{R}, \mathbf{y})$)
29     end
30     $[D[\mathbb{I}[t-k]]...D[\mathbb{I}[t-1]]] := (\mathbf{X}+\mathbf{R}, \mathbf{y})$
31 end
```

$$\nabla_i(W) = \frac{\partial}{\partial W} L_i \in C^1(\Omega),$$

$$\nabla'_i(W) = \frac{\partial^2}{\partial W^2} L_i \in C^0(\Omega).$$

Then, $||\nabla_i(W)|| < \alpha$ and $||\nabla'_i(W)|| < \beta$.

In Attack III, (Line 22 of Algorithm 5), W'_t is calculated as

$$W'_t = W_{t-k} - \frac{\eta''}{k}(\nabla_1(W_{t-k}) + \nabla_2(W_{t-k}) + ... + \nabla_k(W_{t-k}))$$

Whereas, in PoL verification (Line 29 of Algorithm 2),

$$\begin{aligned}\hat{W}_{t-k+1} &= W_{t-k} - \eta\nabla_1(W_{t-k}) \\ \hat{W}_{t-k+2} &= \hat{W}_{t-k+1} - \eta\nabla_2(\hat{W}_{t-k+1}) \\ &... \\ \hat{W}_t &= \hat{W}_{t-1} - \eta\nabla_k(\hat{W}_{t-1})\end{aligned}$$

It is identical to

$$\hat{W}_t = W_{t-k} - \eta(\nabla_1(W_{t-k}) + \nabla_2(\hat{W}_{t-k+1}) + ... + \nabla_k(\hat{W}_{t-1}))$$

If $\mathcal{A}$ sets $\eta'' = k\eta$, then

$$\begin{aligned}\hat{W}_t - W'_t = \eta[&(\nabla_2(W_{t-k}) - \nabla_2(\hat{W}_{t-k+1}) + \\ &(\nabla_3(W_{t-k}) - \nabla_3(\hat{W}_{t-k+2}) + ... + \\ &(\nabla_k(W_{t-k}) - \nabla_k(\hat{W}_{t-1})]\end{aligned}$$

Assuming $[\hat{W}_{t-k+l}, W_{t-k}] = \{W \in \mathbb{R}^n, W = \hat{W}_{t-k+l} + \theta h, 0 \leq \theta \leq 1\}$ is a closed set, and $\nabla_i(W) \in C^1(\Omega)$. Based on the finite-increment theorem (Chapter 10, Section 4 of [2]), we have

$$\begin{aligned}||\nabla_i(W_{t-k}) - \nabla_i(\hat{W}_{t-k+l})|| &\leq \sup_W ||\frac{\partial \nabla_i(W)}{\partial W}|| \cdot ||h|| \\ &\leq \beta||W_{t-k} - \hat{W}_{t-k+l}||\end{aligned}$$

Given

$$\begin{aligned}||W_{t-k} - \hat{W}_{t-k+l}|| = \eta||&\nabla_1(W_{t-k}) + \nabla_2(\hat{W}_{t-k+1}) + ... \\ &+ \nabla_{l-1}(\hat{W}_{t-k+l-1})|| \\ &\leq (l-1)\eta\alpha,\end{aligned}$$

we have

$$||\nabla_i(W_{t-k}) - \nabla_i(\hat{W}_{t-k+l})|| \leq (l-1)\eta\alpha\beta.$$

Then,

$$\begin{aligned}\hat{W}_t - W'_t &\leq \eta^2\alpha\beta\sum_{l=1}^{k-1}(l-1) \\ &= \eta^2\alpha\beta\frac{(k-1)(k-2)}{2}\end{aligned}$$

Recall that $d(W'_t, W_t) \leq \gamma - \sigma$ (Line 23 in Algorithm 5). Then,

$$\begin{aligned}||\hat{W}_t - W_t|| &= ||\hat{W}_t - W'_t + W'_t - W_t|| \\ &\leq ||\hat{W}_t - W'_t|| + ||W'_t - W_t|| \\ &\leq \eta^2\alpha\beta\frac{(k-1)(k-2)}{2} + \gamma - \sigma\end{aligned}$$

□

Therefore, Attack III can pass the verification if we set $\sigma > \eta^2\alpha\beta\frac{(k-1)(k-2)}{2}$.

IV. Evaluation

In this section, we empirically evaluate our attacks in two metrics:

- **Reproducibility.** We need to show that the normalized reproduction errors (cf. Section II-D) in l_1, l_2, l_∞ and cos introduced by our PoL spoof $\mathcal{P}(\mathcal{A}, f_{W_T})$ are smaller than those introduced by a legitimate PoL proof $\mathcal{P}(\mathcal{T}, f_{W_T})$. That means, as long as $\mathcal{P}(\mathcal{T}, f_{W_T})$ can pass the verification, our spoof $\mathcal{P}(\mathcal{A}, f_{W_T})$ can pass the verification as well.
- **Spoof cost.** Recall that a successful PoL spoof requires $\mathcal{A}$ to spend less computation and storage than $\mathcal{T}$ (cf. Section II-A). Therefore, we need to show that the generation time and size of $\mathcal{P}(\mathcal{A}, f_{W_T})$ are smaller than those of $\mathcal{P}(\mathcal{T}, f_{W_T})$.

A. Setting

Following the experimental setup in [14], we evaluate our attacks for ResNet-20 [13] and ResNet-50 [13] on CIFAR-10 [16] and CIFAR-100 [16] respectively. Each of them contains 50,000 training images and 10,000 testing images; and each image is of size 32×32×3. CIFAR-10 only has 10 classes and CIFAR-100 has 100 classes.

We reproduced the results in [14] as baselines for our attacks. Namely, we generate $\mathcal{P}(\mathcal{T}, f_{W_T})$ by training both models for 200 epochs with 390 steps in each epoch (i.e., $E = 200$, $S = 390$) with batch sizes being 128; we set k as 100[4] and measure the normalized reproduction errors and costs. Since our attacks are stochastic spoofing, where $\mathcal{P}(\mathcal{A}, f_{W_T})$ is *not* required to be exactly the same as $\mathcal{P}(\mathcal{T}, f_{W_T})$, we could use different T, k and batch size in our attacks. Recall that $\mathcal{V}$ only verifies $Q < \frac{S}{k}$ largest updates for each epoch (cf. Algorithm 2). Therefore, we measure the reproducibility and spoof cost for both $Q = \lfloor \frac{S}{k} \rfloor$ (full verification) and $Q = 1$ (top-Q verification).

We set k as 100 and batch size as 10 for all of our attacks except Attack III on CIFAR-100[5]. Since it is difficult for Attack I to converge, we focus on evaluating Attack II and Attack III.

B. Attack II

Figure 2 shows the evaluation results of Attack II on CIFAR-10. The results show that the normalized reproduction errors introduced by $\mathcal{P}(\mathcal{A}, f_{W_T})$ are always smaller than those introduced by $\mathcal{P}(\mathcal{T}, f_{W_T})$ in l_1, l_2 and l_{cos} (Figure 2(a), 2(b) and 2(d)). For l_∞, it requires $T' > 15$ for $\mathcal{P}(\mathcal{A}, f_{W_T})$ to be able to pass the verification (Figure 2(c)). On the other hand, when $T' > 30$, the generation time of $\mathcal{P}(\mathcal{A}, f_{W_T})$ is larger than that of $\mathcal{P}(\mathcal{T}, f_{W_T})$ (Figure 2(e)). That means $15 < T' < 30$ would be the condition for Attack II to be successful on CIFAR-10. Notice that the spoof size is always smaller than the proof size.

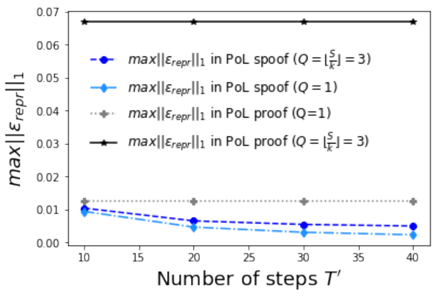
Figure 3 shows the evaluation results of Attack II on CIFAR-100. When $T' > 40$, the normalized reproduction errors introduced by $\mathcal{P}(\mathcal{A}, f_{W_T})$ are smaller than those introduced by $\mathcal{P}(\mathcal{T}, f_{W_T})$ in all 4 distances. On the other hand, it requires $T' < 50$ for the spoof generation time to be smaller than the baseline. Therefore, $40 < T' < 50$ is the condition for Attack II to be successful on CIFAR-100.

C. Attack III

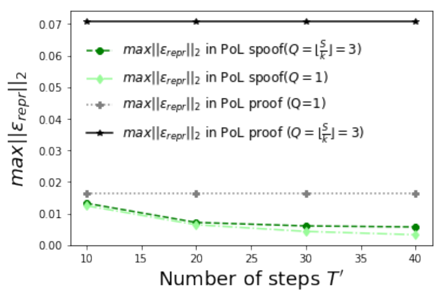
The evaluation results of Attack III on CIFAR-10 are shown in Figure 4. The results show that the normalized reproduction errors introduced by $\mathcal{P}(\mathcal{A}, f_{W_T})$ are always smaller than those introduced by $\mathcal{P}(\mathcal{T}, f_{W_T})$ in all 4 distances when $T' > 25$ (Figure 4(a)-4(d)). In terms of spoof generation time, the number of steps T' can be as large as 300 (Figure 4(e)). That

[4]The authors in [14] suggest to set $k = S$ (which is 390), but in that case Q can only be one.

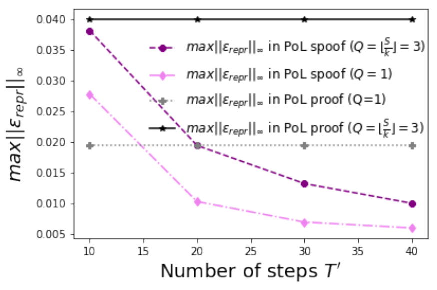
[5]The model for CIFAR-100 is large and Attack III needs to load all k batches into the memory. So we have to use a smaller k or batch size.



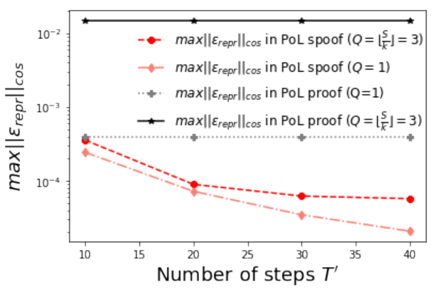
(a) Normalized reproduction error in l_1



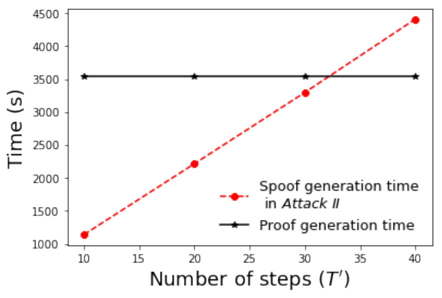
(b) Normalized reproduction error in l_2



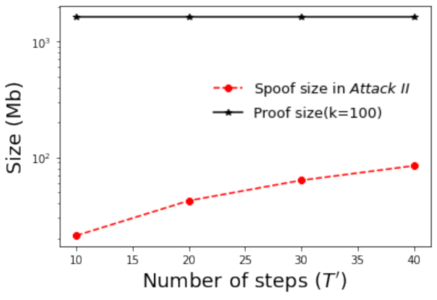
(c) Normalized reproduction error in l_∞



(d) Normalized reproduction error in *cos*

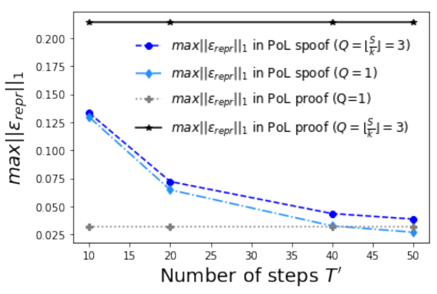


(e) Spoof generation time.

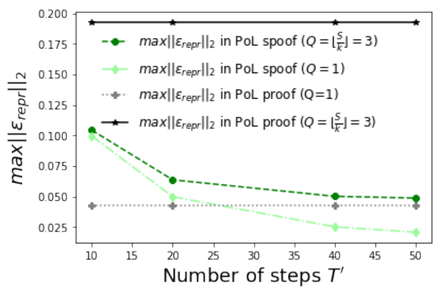


(f) Spoof size.

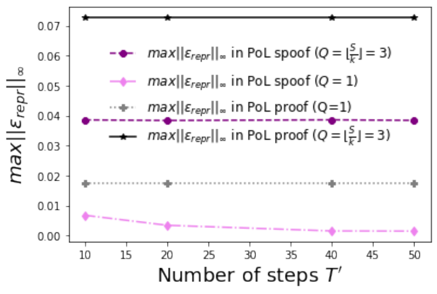
Figure 2. Attack II on CIFAR-10



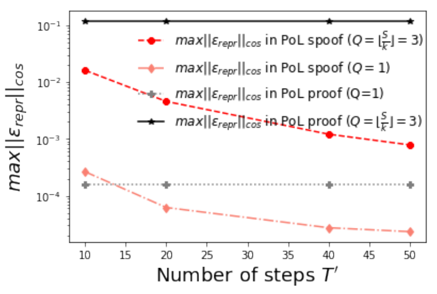
(a) Normalized reproduction error in l_1



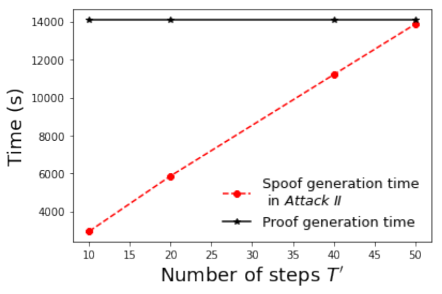
(b) Normalized reproduction error in l_2



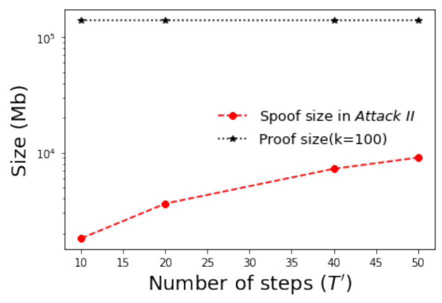
(c) Normalized reproduction error in l_∞



(d) Normalized reproduction error in *cos*

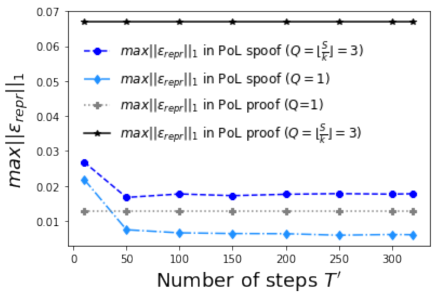


(e) Spoof generation time.

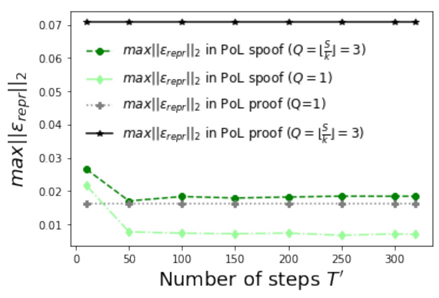


(f) Spoof size.

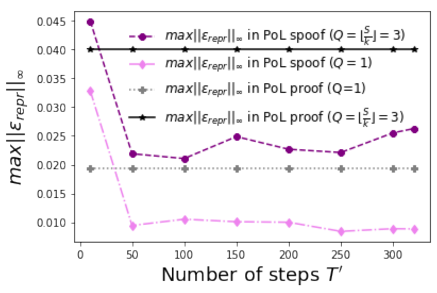
Figure 3. Attack II on CIFAR-100



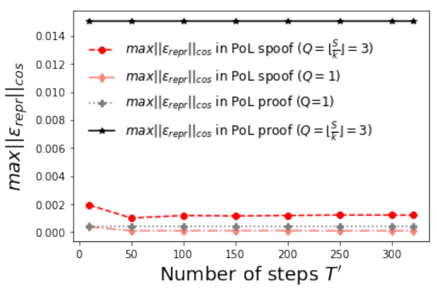
(a) Normalized reproduction error in l_1.



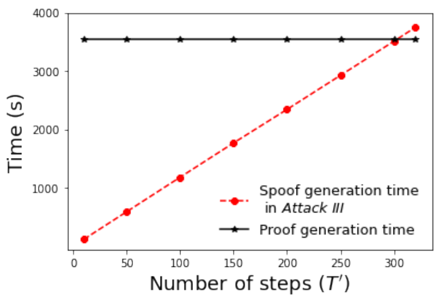
(b) Normalized reproduction error in l_2.



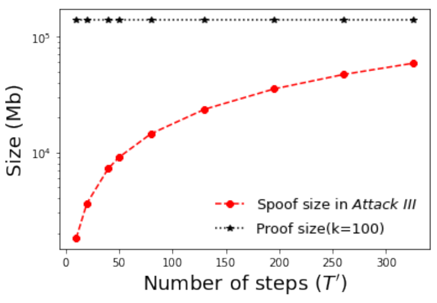
(c) Normalized reproduction error in l_∞.



(d) Normalized reproduction error in cos.

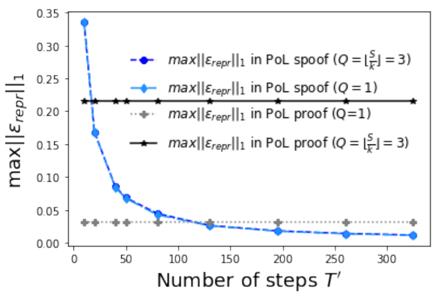


(e) Spoof generation time.

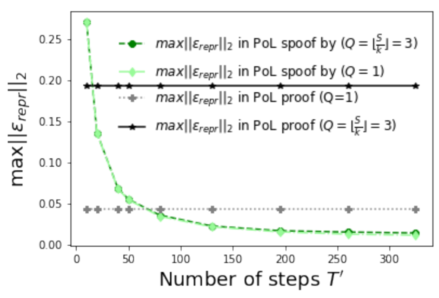


(f) Spoof Size.

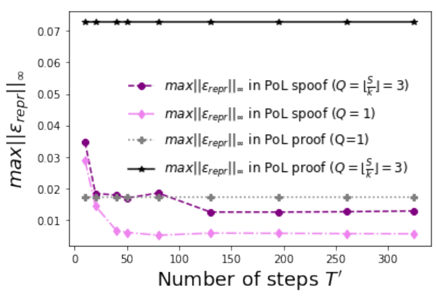
Figure 4. Attack III on CIFAR-10.



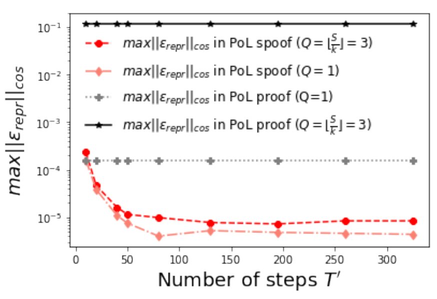
(a) Normalized reproduction error in l_1.



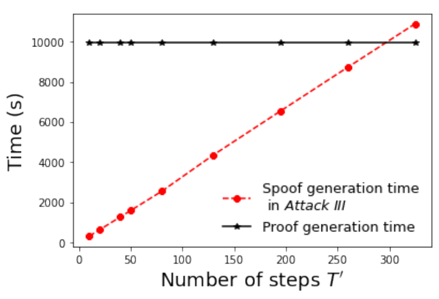
(b) Normalized reproduction error in l_2.



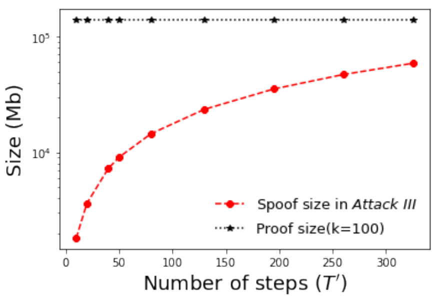
(c) Normalized reproduction error in l_∞.



(d) Normalized reproduction error in cos.

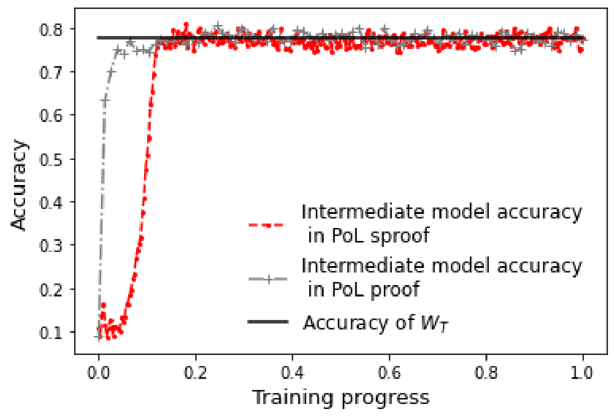


(e) Spoof generation time.

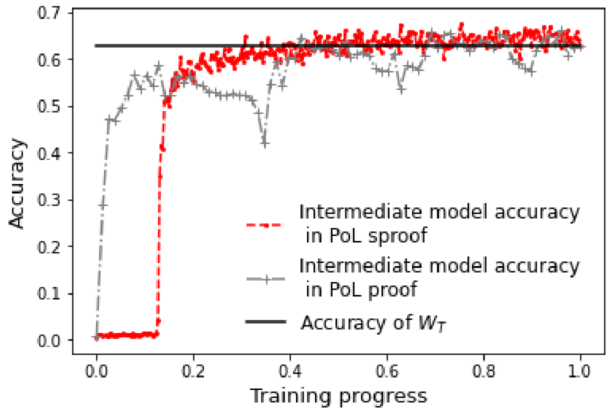


(f) Spoof size.

Figure 5. Attack III on CIFAR-100.

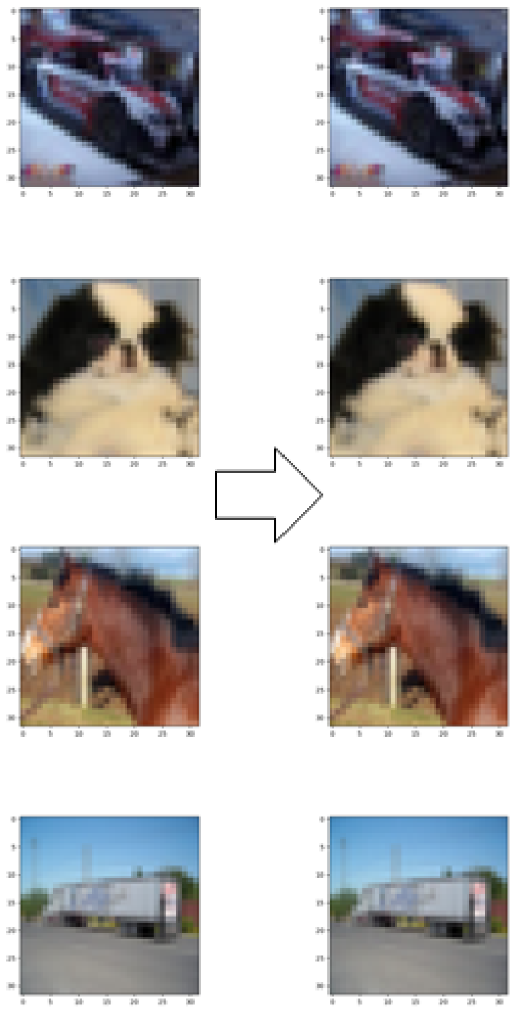


(a) Intermediate models accuracy on CIFAR-10

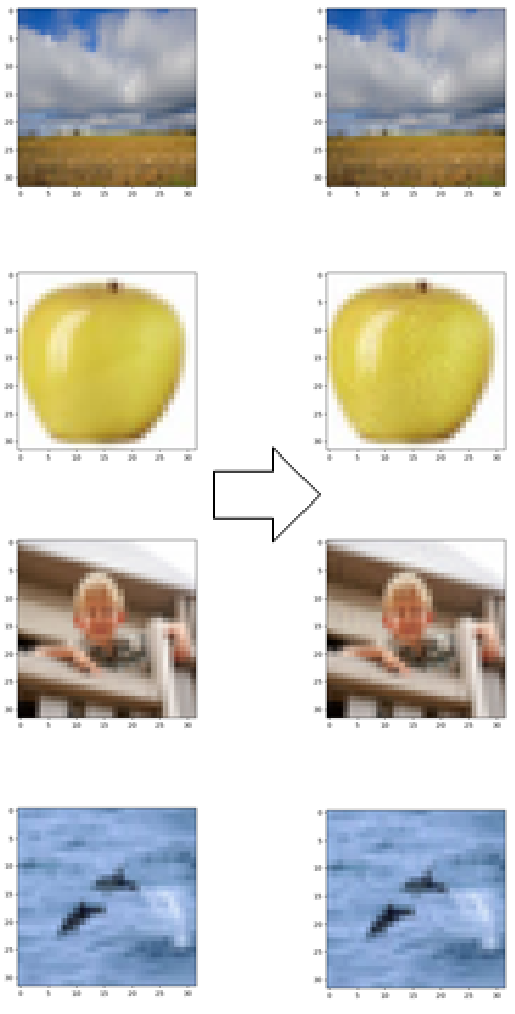


(b) Intermediate models accuracy on CIFAR-100

Figure 6. Intermediate model accuracy ($T' = 300$)



(a) CIFAR-10



(b) CIFAR-100

Figure 7. “Adversarial examples” generated by Attack III. The original images are on the left-hand side and the noised images are on the right-hand side.

means the condition for Attack III to be successful on CIFAR-100 is $25 < T' < 300$.

Figure 5 shows the evaluation results of Attack III on CIFAR-100. We set $k = 100$ and batch size as 2, and evaluate the attack with different number of steps (T'). The results show that the normalized reproduction errors introduced by $\mathcal{P}(\mathcal{A}, f_{W_T})$ are always smaller than those introduced by $\mathcal{P}(\mathcal{T}, f_{W_T})$ in l_∞ and cos (Figure 5(c) and Figure 5(d)). The cross-over point is around 100 in both l_1 and l_2: when $T' > 100$, the normalized reproduction errors introduced by $\mathcal{P}(\mathcal{A}, f_{W_T})$ are smaller than those introduced by $\mathcal{P}(\mathcal{T}, f_{W_T})$ in l_1 and l_2 (Figure 5(a) and Figure 5(b)). Same as on CIFAR-10, the spoof generation time is smaller than the proof generation time when $T' < 300$. Therefore, the condition for Attack III to succeed in CIFAR-100 is $100 < T' < 300$.

In [14], the authors suggest to check model performance periodically. Therefore, we need to make sure that the performance of the intermediate models generated by our attacks follow the same tend as those in $\mathcal{P}(\mathcal{T}, f_{W_T})$. We can achieve this by adjusting the extent of perturbations on W_T (Line 7 in Algorithm 5). Specifically, we can add a large extent of perturbations when T' is small and add a small extent of perturbations when T' is large. Figure 6 shows the model performance in both CIFAR-10 and CIFAR-100. The x-axis presents the progress of training. For example when $x = 0.2$, the corresponding y represents the $0.2 \cdot T$-th model performance in $\mathcal{P}(\mathcal{T}, f_{W_T})$ and $0.2 \cdot T'$-th model performance $\mathcal{P}(\mathcal{A}, f_{W_T})$. It shows that the model performance in $\mathcal{P}(\mathcal{T}, f_{W_T})$ and $\mathcal{P}(\mathcal{A}, f_{W_T})$ are in similar trends.

Figure 7 shows some randomly picked images before and after running our attacks. The differences are invisible.

D. Summary

In summary, both Attack II and Attack III can successfully spoof a PoL when T' is large enough. We further remark that our attacks are slow mostly because they are not well-engineered as normal training, e.g., the optimizers in Tensorflow [1] and Pytorch [26] adopt a bunch of tricks to accelerate training. Therefore, the number of steps T' could be potentially increased, making our attack more imperceptible.

V. Countermeasures

In this section, we provide some potential countermeasures for our attacks. However, even with these countermeasures, we still cannot guarantee a PoL mechanism is secure without a formal proof. Indeed, these countermeasures have their own limitations and can be broken.

Model distance. Recall that Attack II and Attack III require:

$$d(W_t, W_{t-k}) \leq \delta.$$

However, in our experiments, we found that such distances in real training are usually larger. Therefore, $\mathcal{V}$ could set a bound for the distance between each (W_t, W_{t-k}). Notice that this countermeasure is invalid for Attack I.

Integrity of training dataset. In [14], it is assumed that $\mathcal{A}$ has full access to the training dataset D and can modify it. If $\mathcal{A}$ has full access to D, most probably D is a public dataset. Then, $\mathcal{V}$ can easily verify the integrity of D so that $\mathcal{A}$ cannot modify it. If D is a private dataset, then $\mathcal{A}$ can neither access nor modify it. That means we could change the threat model of PoL to make the attack more difficult. However, $\mathcal{A}$ could still spoof W_T using a totally different dataset D' and claim that D' is the real training dataset of W_T.

Checkpointing interval and batch size. Recall that Attack III for CIFAR-100 has to use either a small k or a small batch size due to memory limitation. Then, $\mathcal{V}$ could set bounds for both k and the batch size. However, this countermeasure is invalid for either Attack I or Attack II. When $\mathcal{A}$ has a large GPU memory, it is invalid for Attack III either.

Number of training steps. Recall that Attack II requires $T' < 30$ to succeed and Attack III requires $T' < 300$. Then, $\mathcal{V}$ could set a bound for the number of training steps. However, as we mentioned, our attacks are slow mostly because they are not well-engineered as normal training. We could potentially accelerate our attacks using the tricks adopted in the state-of-the-art optimizers. Then, bounding the number of of training steps will be no longer useful. Furthermore, this countermeasure is invalid for Attack I.

Selection of threshold. As we observed in our experiments, ε_{repr} in the early training stage is usually larger than that in the later stage, because the model converges in the later stage. On the other hand, ε_{repr} remains in the same level in our spoof. Then, it is unreasonable for $\mathcal{V}$ to set a single verification threshold for the whole training stage. A more sophisticated way would be dynamically choosing the verification thresholds according to the stage of model training: choose larger thresholds for the early stage, and choose smaller thresholds for the later stage. In that case, it will be more challenging for our spoof to pass the verification. However, we can also set $d(W_t, W_{t-k})$ closer in the later stage to circumvent this countermeasure.

VI. Related Work

A. Adversarial examples

When first discovered in 2013 [29], adversarial examples are images designed intentionally to cause deep neural networks to make false predictions. Such adversarial examples look almost the same as original images, thus they show vulnerabilities of deep neural networks [11]. Since then, it becomes a popular research topic and has been explored extensively in both attacks [10], [28] and defences [5], [8], [20], [24], [33], [34]. In particular, adversarial examples have been found in many domains other than images, such as autonomous driving [18], [19], malware detection [12], [27], authentication [35] and so on. It becomes a major concern when considering safety issues.

Roughly speaking, adversarial examples are generated by solving:

$$\mathbf{R} = \arg\min_{\mathbf{R}} L(f_W(\mathbf{X} + \mathbf{R}), \mathbf{y}') + \alpha||\mathbf{R}||,$$

where $\mathbf{y}'$ is a label that is different from the real label $\mathbf{y}$ for $\mathbf{X}$. Then, the noise $\mathbf{R}$ can fool the model to predict a wrong label (by minimizing the loss function) and pose little influence on the original instance $\mathbf{X}$.

Recall that the objective of Attack II and Attack III is to minimize the gradients computed by "adversarial examples". Therefore, it can be formulated as:

$$\mathbf{R} = \arg\min_{\mathbf{R}} L(f_W(\mathbf{X}+\mathbf{R}), \mathbf{y}) + \alpha||\mathbf{R}||.$$

This is identical to the objective of finding an adversarial example. An adversarial example aims to fool the model whereas our attacks aim to update a model to itself. Nevertheless, they end up at the same point, which explains the effectiveness of Attack II and Attack III.

Goodfellow et al. [11] proposed a one-step method called fast gradient sign method (FGSM) to generate adversarial examples:

$$\mathbf{X} = \mathbf{X} + \epsilon sign(\triangledown_{\mathbf{X}} L(f_W(\mathbf{X}), \mathbf{y})),$$

where $\epsilon sign(\triangledown_{\mathbf{X}} L(f_W(\mathbf{X}), \mathbf{y}))$ is the max-norm constrained perturbation and it can be computed by one step back-propagation. Therefore, FGSM is much more efficient.

Su et al. [28] target a limited scenario where only one pixel can be modified. They use l_0 norm as a constraint $||e(\mathbf{x})||_0 \leq d$ to modify limited pixels. In their experiments, they choose $d = 1$ so that only one pixel can be modified. Then, the objective function is:

$$\arg\max_{e(\mathbf{X})} L(f(\mathbf{X}+e(\mathbf{X})), \mathbf{y}).$$

Their work show that more than 60% of the CIFAR-10 images can be perturbed by their method.

We leave it as future work to estimate the effectiveness of these works in our settings.

B. Deep Leakage from Gradient

In federated learning [7], [15], [17], [21]), it was widely believed that shared gradients will not leak information about the training data. However, Zhu et al. [36] proposed "Deep Leakage from Gradients" (DLG), where the training data can be recovered through gradients matching. Specifically, after receiving gradients from another worker, the adversary feeds a pair of randomly initialized dummy instance (X, y) into the model, and obtains the dummy gradients via back-propagation. Then, they update the dummy instance with an objective of minimizing the distance between the dummy gradients and the received gradients:

$$\begin{aligned}\mathbf{X}, y &= \arg\min_{\mathbf{X}, y} ||\triangledown W' - \triangledown W||^2 \\ &= \arg\min_{\mathbf{X}, y} ||\frac{\partial L(f_W(\mathbf{X}), y)}{\partial W} - \triangledown W||^2\end{aligned}$$

After a certain number of steps, the dummy instance can be recovered to the training data of the worker who sent the gradients.

Our attacks were largely inspired by DLG: an instance can be updated so that its output gradients can match the given gradients. The difference between DLG and our work is that DLG aims to recover the training data from the gradients, whereas we want to create a perturbation on a real instance to generate specific gradients.

C. Model stealing

One goal of PoL is to resolve the dispute caused by model stealing attacks, which allow adversaries to learn a new model $\hat{f}$ that is close to the target model f. Tramer et al. [30] proposed the first model stealing attack that leverages the outputs from the target model. They demonstrate that for a large variety of machine learning models, their model extraction attacks can successfully "steal" the weights of the target model. Orekondy et al. [23] proposed a two-step model stealing attack. By querying data and using the output returned by the target model, they train a "knockoff" model F_A to approximate the target model. They show that their "knockoff" model achieves about 80% accuracy of the target model.

Besides model weights, hyperparameters are also important for a machine learning model. Wang et al. [32] came up with an attack for stealing hyperparameters. Given access to training data, objective function and model weights, they use a linear square method to solve the overdetermined linear system of the hyperparameters. There are also some work to attack the architecture and the optimization process of the target model [22] under the assumption that adversary can only access to the query inputs and outputs.

D. Byzantine workers

Another goal of PoL is to prevent Byzantine workers from conducting denial-of-service attacks in federated learning. A Byzantine worker can manipulate its gradients to prevent the convergence of the global model. Blanchard et al. [6] prove that no aggregation rule based on linear combination (e.g., average) of the updates can tolerate a single Byzantine participant. As a result, most researchers working on this direction are exploring alternative aggregation rules. For example, in the Krum aggregation rule [6], pairwise distances are calculated between all inputs submitted in an iteration, and picks the one with the lowest sum as the aggregated gradients. Unfortunately, this line of work assumes that participants' training data are i.i.d., i.e., following the same distribution.

E. Verifiable computation

In general, verifiable computation allows a delegator to outsource the execution of a complex function to some workers, and the delegator verifies the correctness of the returned result while performing less work than executing the function itself. A verifiable computation system for an NP relationship R is a protocol between a computationally bounded prover and a verifier. At the end of the protocol, the verifier is convinced by prover that there exists a witness w such that $(x; w) \in R$ for some input x. The correctness property of verifiable computation guarantees that an honest prover can always pass the verification, and the soundness guarantees that a cheating prover will be caught with overwhelming probability. There are many solutions for verifiable computation such as SNARK [4], [25], STARK [3] etc. We refer to [31] for a complete survey.

PoL is definitely a problem of verifiable computation. We leave it as future work to investigate how to design a secure and efficient PoL mechanism using verifiable computation.

VII. Conclusion

In this paper, we show that a recently proposed PoL mechanism is vulnerable to "adversarial examples". Namely, in a similar way as generating adversarial examples, we could generate a PoL spoof with significantly less cost than generating a proof by the prover. We validated our attacks by conducting experiments extensively. In future work, we will explore more effective attacks. We will also design a PoL mechanism that is both secure and efficient.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[2] Vladimir A. Zorich (auth.). *Mathematical Analysis II*. Universitext. Springer, 2nd edition, 2016.

[3] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint, 2018.*

[4] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *Proceedings of the USENIX Security Symposium, 2014*.

[5] Arjun Nitin Bhagoji, Daniel Cullina, Chawin Sitawarin, and Prateek Mittal. Enhancing robustness of machine learning systems via data transformations. In *2018 52nd Annual Conference on Information Sciences and Systems (CISS)*, pages 1–5, 2018.

[6] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. Machine learning with adversaries: Byzantine tolerant gradient descent. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[7] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečnỳ, Stefano Mazzocchi, H Brendan McMahan, et al. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046*, 2019.

[8] Jacob Buckman, Aurko Roy, Colin Raffel, and Ian Goodfellow. Thermometer encoding: One hot way to resist adversarial examples. In *International Conference on Learning Representations*, 2018.

[9] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning, 2014.

[10] Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, and Jianguo Li. Boosting adversarial attacks with momentum. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 9185–9193, 2018.

[11] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[12] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick D. McDaniel. Adversarial examples for malware detection. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II*, volume 10493 of *Lecture Notes in Computer Science*, pages 62–79. Springer, 2017.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[14] Hengrui Jia, Mohammad Yaghini, Christopher A. Choquette-Choo, Natalie Dullerud, Anvith Thudi, Varun Chandrasekaran, and Nicolas Papernot. Proof-of-learning: Definitions and practice. In *IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2021.

[15] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *arXiv preprint arXiv:1912.04977*, 2019.

[16] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[17] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. Federated learning: Challenges, methods, and future directions. *IEEE Signal Processing Magazine*, 37(3):50–60, 2020.

[18] Jiajun Lu, Hussein Sibai, and Evan Fabry. Adversarial examples that fool detectors, 2017.

[19] Jiajun Lu, Hussein Sibai, Evan Fabry, and David Forsyth. No need to worry about adversarial examples in object detection in autonomous vehicles, 2017.

[20] Yan Luo, Xavier Boix, Gemma Roig, Tomaso Poggio, and Qi Zhao. Foveation-based mechanisms alleviate adversarial examples, 2016.

[21] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In Aarti Singh and Jerry Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54, pages 1273–1282, Fort Lauderdale, FL, USA, 20–22 Apr 2017. PMLR.

[22] Seong Joon Oh, Bernt Schiele, and Mario Fritz. Towards reverse-engineering black-box neural networks. In *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, pages 121–144. Springer, 2019.

[23] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Knockoff nets: Stealing functionality of black-box models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4954–4963, 2019.

[24] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *2016 IEEE symposium on security and privacy (SP)*, pages 582–597. IEEE, 2016.

[25] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252, May 2013.

[26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[27] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. Generic black-box end-to-end attack against state of the art API call based malware classifiers. In Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis, editors, *Research in Attacks, Intrusions, and Defenses - 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings*, volume 11050 of *Lecture Notes in Computer Science*, pages 490–510. Springer, 2018.

[28] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. One pixel attack for fooling deep neural networks. *IEEE Transactions on Evolutionary Computation*, 23(5):828–841, 2019.

[29] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[30] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 601–618, 2016.

[31] Michael Walfish and Andrew J Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2):74–84, 2015.

[32] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 36–52. IEEE, 2018.

[33] Qinglong Wang, Wenbo Guo, Kaixuan Zhang, Alexander G. Ororbia II au2, Xinyu Xing, Xue Liu, and C. Lee Giles. Learning adversary-resistant deep neural networks, 2017.

[34] Stephan Zheng, Yang Song, Thomas Leung, and Ian Goodfellow. Improving the robustness of deep neural networks via stability training. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4480–4488, 2016.

[35] Zhe Zhou, Di Tang, Xiaofeng Wang, Weili Han, Xiangyu Liu, and Kehuan Zhang. Invisible mask: Practical attacks on face recognition with infrared. *arXiv preprint arXiv:1803.04683*, 2018.

[36] Ligeng Zhu and Song Han. Deep leakage from gradients. In *Federated learning*, pages 17–31. Springer, 2020.