SplitGuard: Detecting and Mitigating Training-Hijacking Attacks in Split Learning

Ege Erdoğan
Dept. of Computer Engineering
Koç University
Istanbul, Turkey
eerdogan17@ku.edu.tr

Alptekin Küpçü
Dept. of Computer Engineering
Koç University
Istanbul, Turkey
akupcu@ku.edu.tr

A. Ercüment Çiçek
Dept. of Computer Engineering
Bilkent University
Ankara, Turkey
cicek@cs.bilkent.edu.tr

***Abstract*—Distributed deep learning frameworks, such as *split learning*, have recently been proposed to enable a group of participants to collaboratively train a deep neural network without sharing their raw data. Split learning in particular achieves this goal by dividing a neural network between a client and a server so that the client computes the initial set of layers, and the server computes the rest. However, this method introduces a unique attack vector for a malicious server attempting to steal the client's private data: the server can direct the client model towards learning a task of its choice. With a concrete example already proposed, such *training-hijacking* attacks present a significant risk for the data privacy of split learning clients.**

In this paper, we propose SplitGuard, a method by which a split learning client can detect whether it is being targeted by a training-hijacking attack or not. We experimentally evaluate its effectiveness, and discuss in detail various points related to its use. We conclude that SplitGuard can effectively detect training-hijacking attacks while minimizing the amount of information recovered by the adversaries.

***Index Terms*—machine learning, data privacy, split learning**

I. Introduction

As neural networks, and more specifically *deep neural networks* (DNNs), began outperforming traditional machine learning methods in tasks such as natural language processing [1], they became the workhorses driving the field of machine learning forward. However, effectively training a DNN requires large amounts of computational power and high-quality data [2]. On the other hand, relying on a sustained increase in computing power is unsustainable [3], and it may not be possible to share data freely in fields such as healthcare [4], [5].

To alleviate these two problems, distributed deep learning methods such as *split learning* (SplitNN) [6], [7] and *federated learning* (FL) [8]–[10] have been proposed. They fulfill their purpose by enabling a group of data-holders to collaboratively train a DNN without sharing their private data, while offloading some of the computational work to a more powerful server.

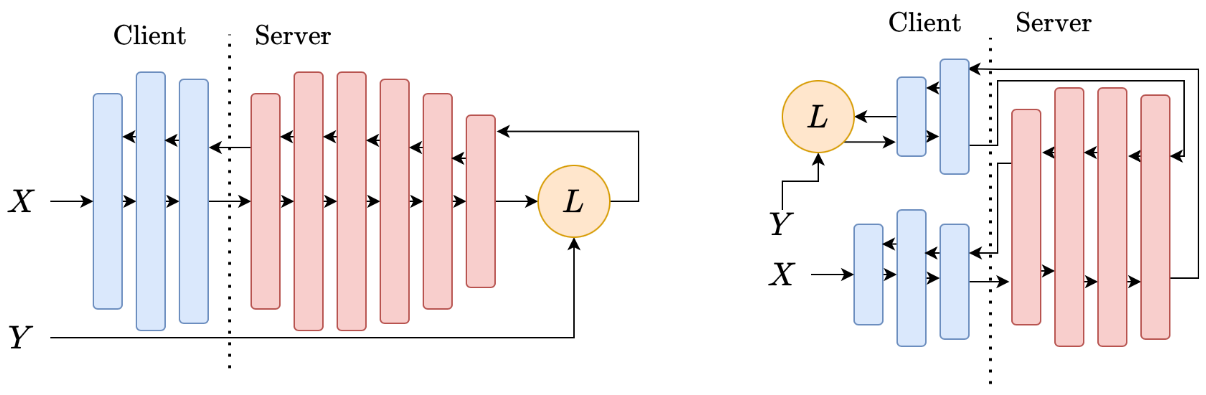
In FL, each client trains a DNN using its local data, and sends its parameter updates to the central server; the server then aggregates the updates in some way (e.g. average) and sends the aggregated results back to each client. In SplitNN, a DNN is split into two parts and the clients train in a round-robin manner. The client taking its turn computes the first few layers of the DNN and sends the output to the server, who then computes the DNN's overall output and starts the parameter updates by calculating the loss value. In both methods, no client shares its private data with another party, and all clients end up with the same model.

Motivation. In SplitNN, the server has control over the parameter updates being propagated back to each client model. This creates a new attack vector, that has already been exploited in an attack proposed by Pasquini et al. [11], for a malicious server trying to infer the clients' private data. By contrast, this attack vector does not exist in federated learning, since the clients can trivially check if their model is aligned with their goals by calculating its accuracy. The same process is not possible in split learning, since the adversary can train a legitimate model on the side using the clients' intermediate outputs, and use that model for a performance measure. In fact, any such detection protocol that expects cooperation from the server is doomed to failure through the server's use of a legitimate surrogate model as described.

Contributions. Our main contribution in this paper is SplitGuard, a protocol by which a SplitNN client can detect, without expecting cooperation from the server, if its local model is being hijacked. To the best of our knowledge, SplitGuard is the first attempt at detecting training-hijacking attacks against split learning clients. To achieve our goal, we utilize the observation that if a client's local model is learning the intended task, then it should behave in a drastically different way when the task is reversed (i.e. when success in the original task implies failure in the new task). We demonstrate using three commonly used benchmark datasets (MNIST [12], Fashion-MNIST [13], and CIFAR10 [14]) that SplitGuard effectively detects and mitigates the only training-hijacking attack proposed so far [11]. We further argue that it is generalizable to any such training-hijacking attack.

In the rest of the paper, we first provide the necessary background on DNNs and SplitNN, and explain some of the related work. We then describe SplitGuard, experimentally evaluate it, and discuss certain points pertaining to its use. We conclude by providing an outline of possible future work related to SplitGuard.

Supplementary code for the paper can be found at `https://github.com/ege-erdogan/splitguard`.



(a) With label-sharing.

(b) Without label-sharing.

Fig. 1: Two different split learning setups. Arrows denote the forward and backward passes, starting with the examples X, and propagating backwards after the loss computation using the labels Y. In Figure 1a, clients send the labels to the server along with the intermediate outputs. In Figure 1b, the model terminates on the client side, and thus the clients do not have to share their labels with the server.

II. Background

A. Neural Networks

A neural network [15] is a parameterized function $f : X \times \Theta \rightarrow Y$ that tries to approximate a function $f^* : X \rightarrow Y$. The goal of the training procedure is to learn the parameters Θ using a training set consisting of examples $\tilde{X}$ and labels $\tilde{Y}$ sampled from the real-world distributions X and Y.

A typical neural network, also called a *feedforward neural network*, consists of discrete units called *neurons*, organized into layers. Each neuron in a layer takes in a weighted sum of the previous layer's neurons' outputs, applies a non-linear activation function, and outputs the result. The weights connecting the layers to each other constitute the parameters that are updated during training. Considering each layer as a seperate function, we can model a neural network as a chain of functions, and represent it as $f(x) = f^{(N)}(...(f^{(2)}(f^{(1)}(x)))$, where $f^{(1)}$ corresponds to the first layer, $f^{(2)}$ to the second layer, and $f^{(N)}$ to the final, or the *output* layer.

Like many other machine learning methods, training a neural network involves minimizing a loss function. However, since the nonlinearity introduced by the activation functions applied at each neuron causes the loss function to become non-convex, we use iterative, gradient-based approaches to minimize the loss function. It is important to note that these methods do not provide any global convergence guarantees.

A widely-used optimization method is *stochastic gradient descent* (SGD). Rather than computing the gradient from the entire data set, SGD computes gradients for batches selected from the data set. The weights are updated by propagating the error backwards using the backpropagation algorithm. Training a deep neural network generally requires multiple passes over the entire data set, each such pass being called an *epoch*. One round of training a neural network requires two passes through the network: one forward pass to compute the network's output, and one backward pass to update the weights. We will use the terms *forward pass* and *backward pass* to refer to these operations in the following sections. For an overview of gradient-based optimization methods other than SGD, we refer the reader to [16].

B. Split Learning

In split learning (SplitNN) [6], [7], [17], a DNN is split between the clients and a server such that the clients compute the first few layers, and the server computes rest of the layers. This way, a group of clients can train a DNN utilizing, but not sharing, their collective data. Furthermore, most of the computational work is also offloaded to the server, reducing the training cost for the clients. However, this partitioning involves a privacy/cost trade-off for the clients, with the outputs of earlier layers leaking more information about the inputs.

Figure 1 displays the two basic modes of SplitNN, the main difference between the two being whether the clients share their labels with the server or not. In Figure 1a, clients compute only the first few layers, and should share their labels with the server. The server then computes the loss value, starts backpropagation, and sends the gradients of its first layer back to the client, who then completes the backward pass. The private-label scenario depicted in Figure 1b follows the same procedure, with an additional communication step. Since now the client computes the loss value and initiates backpropagation, it should first feed the server model with the gradient values to resume backpropagation.

For our purposes, it is important to realize that the server can launch a training-hijacking attack even in the private-label scenario (Figure 1b). It simply discards the gradients it received from the second part of the client model, and computes a malicious loss function using the intermediate output it received from the first client model, propagating the malicious loss back to the first client model.

The primary advantage of SplitNN compared to federated learning is its lower communication load [18]. While federated learning clients have to share their entire parameter updates with the server, SplitNN clients only share the output of a single layer. However, choosing an appropriate split depth is crucial for SplitNN to actually provide data privacy. If the initial client model is too shallow, an honest-but-curious server can recover the private inputs with high accuracy, knowing only the model architecture (not the parameters) on the clients' side [19]. This implies that SplitNN clients should increase their computational load, by computing more layers, for stronger data privacy.

Finally, SplitNN follows a round-robin training protocol to accomodate multiple clients; clients take turn training with the server using their local data. Before a client starts its turn, it should bring its parameters up to date with those of the most recently trained client. There are two ways to achieve this: the clients can either share their parameters through a central parameter server, or directly communicate with each other in a P2P way and update their parameters.

III. Related Work

A. Feature-Space Hijacking Attack (FSHA)

The Feature-Space Hijacking Attack (FSHA), by Pasquini et al. [11], is the only proposed training-hijacking attack against SplitNN clients so far. It is important to gain an understanding of how a training-hijacking attack might work before discussing SplitGuard in detail.

In FSHA, the atttacker (SplitNN server) first trains an autoencoder (consisting of the encoder $\tilde{f}$ and the decoder $\tilde{f}^{-1}$) on some public dataset X_{pub} similar to that of the client's private dataset X_{priv}. It is important for the attack's effectiveness that X_{pub} be similar to X_{priv}. Without such a dataset at all, the attack cannot be launched. The main idea then is for the server to bring the output spaces of the client model f and the encoder $\tilde{f}$ as close as possible, so that the decoder $\tilde{f}^{-1}$ can successfully invert the client outputs and recover the private inputs.

After this initial *setup phase*, the client model's training begins. For this step, the attacker initializes a distinguisher model D that tries to distinguish the client's output $f(X_{priv})$ from the encoder's output $\tilde{f}(X_{pub})$. More formally, the distinguisher is updated at each iteration to minimize the loss function

$$L_D = \log(1 - D(\tilde{f}(X_{pub}))) + \log(D(f(X_{priv}))). \quad (1)$$

Simultaneously at each training iteration, the server directs the client model f towards maximizing the distinguisher's error rate, thus minimizing the loss function

$$L_f = \log(1 - D(f(X_{priv}))). \quad (2)$$

In the end, the output spaces of the client model and the server's encoder are expected to overlap to a great extent, making it possible for the decoder to invert the client's outputs.

Notice that the client's loss function L_f is totally independent of the training labels, as in changing the value of the labels does not affect the loss function. We will soon refer to this observation.

IV. SplitGuard

We start our presentation of SplitGuard by restating an earlier remark: *If the training-hijacking detection protocol requires the attacker SplitNN server to knowingly take part in the protocol, the server can easily circumvent the protocol by training a legitimate model on the side, and using that model during the protocol's run.* In the light of this, it is evident that we need a method which the clients can run during training, without breaking the flow of training from the server's point of view.

A. Overview

During training with SplitGuard, clients intermittently input batches with randomized labels, denoted *fake batches*. The main idea is that if the client model is learning the intended task, then the gradient values received from the server should be noticeably different for fake batches and *regular batches*.[1]

The client model learning the intended task means that it is moving towards a relatively high-accuracy point on its parameter space. That same high-accuracy point becomes a low-accuracy point when the labels are randomized. The model tries to get away from that point, and the classification error increases. More specifically, we make the following two claims (experimentally validated in Section V-B):

Claim 1. *If the client model is learning the intended task, then the angle between fake and regular gradients will be higher than the angle between two random subsets of regular gradients.*[2]

Claim 2. *If the client model is learning the intended task, then fake gradients will have a higher magnitude than regular gradients.*

Notation	
P_F	Probability of sending a fake batch
B_F	Share of randomized labels in a fake batch
N	Batch index at which SplitGuard starts running
F	Set of fake gradients
R_1, R_2	Random, disjoint subsets of regular gradients
R	$R_1 \cup R_2$
α, β	Parameters of the SplitGuard score function
L	Number of classes
A	Model's classification accuracy
A_F	Expected classification accuracy for a fake batch

TABLE I: Summary of notation used throughout the paper.

B. Putting the Claims to Use

At the core of SplitGuard, clients compute a value, denoted the SplitGuard score, based on the fake and regular gradients

[1] *Fake gradients* and *regular gradients* similarly refer to the gradients resulting from fake and regular batches.

[2] Angle between *sets* meaning the angle between the *sums* of vectors in those sets.

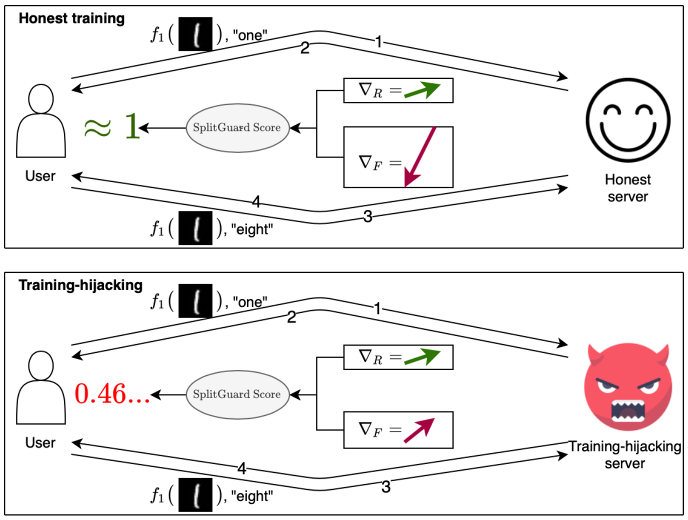


Fig. 2: Overview of SplitGuard, comparing the honest training and training-hijacking scenarios. Clients intermittently send training batches with randomized labels, and analyze the behavior of their local models visible from their parameter updates.

they have collected up to that point. This value's history is then used to reach a decision on whether the server is launching an attack or not. We now describe this calculation process in more detail. Table I displays the notation we use from here on.

Starting with the Nth batch during the first epoch of training, with probability P_F,[3] clients send fake batches with the share $B_F \in [0, 1]$ of the labels randomized. Upon calculating the gradient values for their first layer, clients append the fake gradients to the list F, and split the regular gradients randomly into the lists R_1 and R_2, where $R = R_1 \cup R_2$. To minimize the effect of fake batches on model performance, clients discard the parameter updates resulting from fake batches. Figure 2 displays a simplified overview of the protocol, and Algorithm 1 explains the modified training procedure in more detail. The `MAKE_DECISION` function contains the clients' decison-making logic and will be described later in Algorithm 3.

To make sure that none of the randomized labels gets mapped to its original value, it is a good idea to add to each label a random positive integer between 1 and L exclusive, and compute the result modulo L, where L is the number of classes.

We should first define two quantities. For two sets of vectors A and B, we define $d(A, B)$ as the absolute difference between the average magnitudes of the vectors in A and B:

$$d(A, B) = \left| \frac{1}{|A|} \sum_{a \in A} \|a\| - \frac{1}{|B|} \sum_{b \in B} \|b\| \right|, \qquad (3)$$

[3]This is equivalent to allocating a certain share of the training dataset for this purpose before training.

Algorithm 1: Client training with label sharing

```
$f, w$: client model, parameters
$OPT$: optimizer
$P_F$: probability of sending fake batches
$B_F$: number of labels randomized in fake batches
$N$: number of initial batches to ignore
initialize $R_1, R_2, F$ as empty lists
rand($y$, $B_F$): randomize share $B_F$ of the labels $Y$.
while training do
    for $(x_i, y_i) \leftarrow$ trainset do
        if probability $P_F$ occurs and $i \geq N$ then
            // sending fake batches
            Send $(f(x_i), \mathtt{rand}(y_i, B_F))$ to server
            Receive gradients $\nabla_F$ from server
            Append $\nabla_F$ to F
            MAKE_DECISION($F, R_1 \cup R_2$)
            // do not update parameters
        else
            // regular training
            Send $(f(x_i), y_i)$ to server
            Receive gradients $\nabla_R$ from server
            if $i \geq N$ then
                if probability 0.5 occurs then
                    Append $\nabla_R$ to $R_1$
                else
                    Append $\nabla_R$ to $R_2$
            $w \leftarrow w + OPT(\nabla_R)$
```

and $\theta(A, B)$ as the angle between sums of vectors in two sets A and B:

$$\theta(A, B) = arccos(\frac{\bar{A} \cdot \bar{B}}{\|\bar{A}\| \cdot \|\bar{B}\|}) \qquad (4)$$

where

$$\bar{A} = \sum_{a \in A} a \qquad (5)$$

for a set of vectors A.

Going back to the two claims, we can restate them using these quantities:

Claim 1 restated. $\theta(F, R) > \theta(R_1, R_2)$

Claim 2 restated. $d(F, R) > d(R_1, R_2)$

If the model is learning the intended task, then it follows from the two claims that the product $\theta(F, R) \cdot d(F, R)$ will be greater than the product $\theta(R_1, R_2) \cdot d(R_1, R_2)$. If the model is learning some other task independent of the labels, then F, R_1, and R_2 will essentially be three random samples of the set of gradients obtained during training, and it will not be possible to consistently detect the same relationships among them.

We can now define the values clients compute to reach a decision. First, after each fake batch, the clients compute the value:

$$S = \frac{\theta(F, R) \cdot d(F, R) - \theta(R_1, R_2) \cdot d(R_1, R_2)}{d(F, R) + d(R_1, R_2) + \varepsilon}. \qquad (6)$$

As stated, the numerator contains the useful information we want to extract, and we divide that result by $d(F, R) + d(R_1, R_2) + \varepsilon$, where ε is a small constant to avoid division by zero. This division bounds the S value within the interval $[-\pi, \pi]$, a feature that will shortly come handy.

So far, the claims lead us to consider high S values as indicating an honest server, and low S values as indicating a malicious server. However, the S values obtained during honest training vary from one model/task to another. For a more effective method, we need to define the notions of *higher* and *lower* more clearly. For this purpose, we will define a *squashing function* that maps the interval $[-\pi, \pi]$ to the interval $(0, 1)$, where high S values get mapped infinitesimally close to 1 while the lower values get mapped to considerably lower values.[4] This allows the clients to choose a threshold, such as 0.9, to separate high and low values.

Our function of choice for the squashing function is the logistic sigmoid function σ. To provide some form of flexibility to the clients, we introduce two parameters, α and β, and define the function as follows:

$$SG = \sigma(\alpha \cdot S)^{\beta} \in (0, 1). \qquad \text{(SplitGuard Score)}$$

The function fits naturally for our purposes into the interval $[-\pi, \pi]$, mapping the high-end of the interval to 1, and the lower-end to 0. The parameter α determines the range of values that get mapped very close to 1, while increasing the parameter β punishes the values that are less than 1. We discuss the process of choosing these parameters in more depth in Section VI.

V. Experimental Evaluation

We need to answer three questions to claim that SplitGuard is an effective method:

- How much does sending fake batches affect model performance? If the decrease is significant, then the harm might outweigh the benefit.
- Do the underlying claims hold?
- Can SplitGuard succeed in detecting FSHA, while not reporting an attack during honest training?
- What can a typical adversary learn until detection?

In each of the following subsections, we answer one of these questions by conducting various experiments. For our experiments, we used the ResNet architecture [20], trained with the Adam optimizer [21], on the MNIST [12], Fashion-MNIST [13], and CIFAR10 [14] datasets. We implemented our attack in Python (v 3.7) using the PyTorch library (v 1.9) [22]. In all our experiments, we limit our scope only to the first epoch of training. It is the least favorable time for detecting an attack since the model initially behaves randomly, and represents a lower bound for results in later epochs.

A. Effect on Model Performance

Table II displays the classification accuracy of the ResNet model on the test sets of our three benchmark datasets with different B_F values, averaged over 10 runs. The client model consists of a single convolutional layer, and the rest of the model is computed by the server. This is the worst-case scenario for this purpose, since the part of the model that is being updated with fake batches is as large as possible. Also remember that a B_F value of 1 does not mean that the clients always send fake labels. They are still sending fake labels with probability P_F.

B_F	Classification Accuracy (%)		
	MNIST	**F-MNIST**	**CIFAR**
0 (Original)	97.52	87.77	50.39
1/64	97.72	86.94	51.28
4/64	97.44	87.76	50.39
8/64	97.78	87.33	50.49
16/64	97.70	87.67	52.30
32/64	97.83	87.46	53.68
64/64	97.29	86.30	50.42

TABLE II: Test classification accuracy values of the ResNet model for MNIST, F-MNIST, and CIFAR datasets for different B_F values after the first epoch of training with SplitGuard, averaged over 10 runs with a P_F of 0.1.

The results demonstrate that even when limited to the first epoch, the model performs similarly when trained with and without SplitGuard. There is not a noticeable and consistent decrease in performance for any of the datasets, even for high B_F values such as 1.

B. Validating the Claims

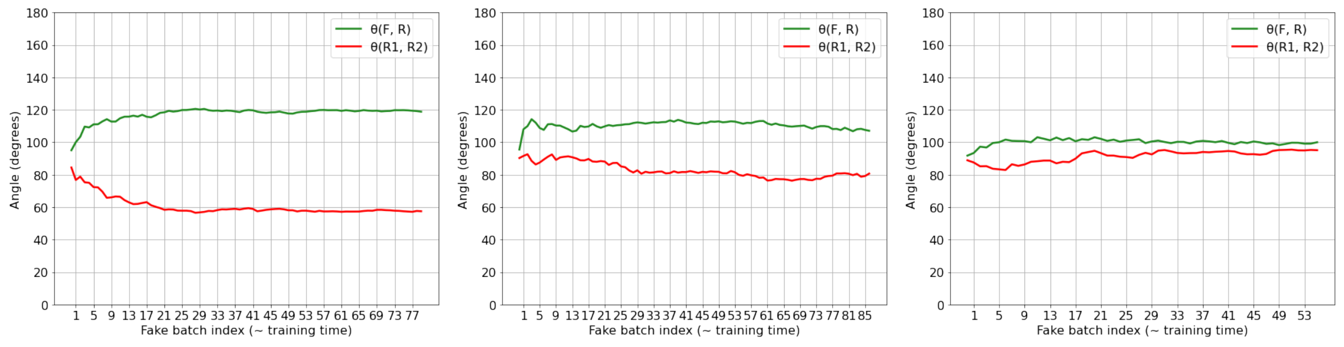
Going back to the two claims, we now demonstrate that fake gradients make a larger angle with regular gradients than the angle between two subsets of regular gradients, and that fake gradients have a higher magnitude than regular gradients. Figures 3 and 4 display these values for each of our three datasets obtained during the first epoch of training with an honest server, averaged over 5 runs.

From Figure 3, it can be observed that $\theta(F, R)$ is consistently greater than $\theta(R_1, R_2)$ for each of our benchmark datasets. Note however that the difference is greater for MNIST (around 60°) than for Fashion-MNIST (around 30°) and CIFAR (around 10°). Remembering from Table II that the model's performance after the first epoch of training is higher in MNIST compared to other datasets, it is not surprising that the difference between the angles is higher as well. As we will discuss later, SplitGuard is more effective as the model becomes more accurate.

Finally, Figure 4 displays a similar relation between the $d(F, R)$ and $d(R_1, R_2)$ values obtained during the first epoch of training. For each of our datasets, $d(F, R)$ values are consistently higher than the $d(R_1, R_2)$ values, although the difference is smaller for CIFAR compared to MNIST.

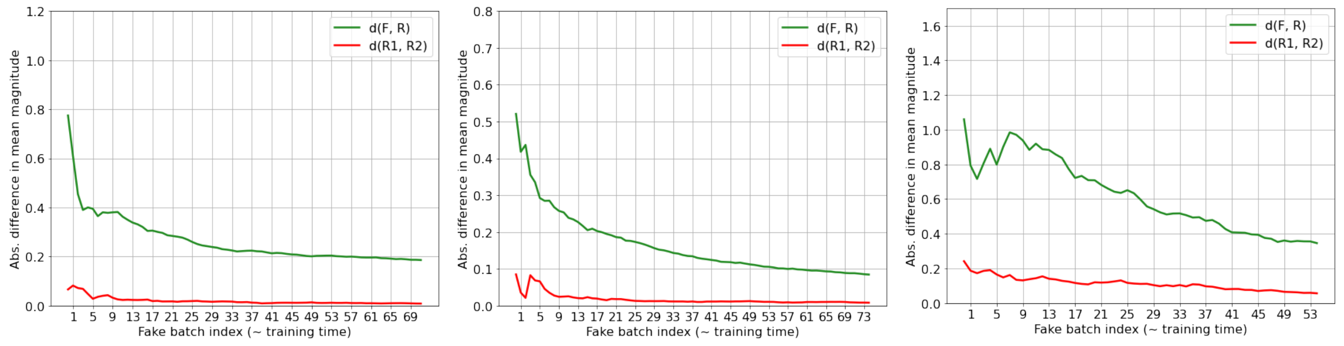
To recap, Figures 3 and 4 demonstrate that our claims are valid during the first epoch of training for our benchmark datasets. The decreasing difference as the models become less adept (going from MNIST to CIFAR10) implies that the

[4]From here on we will refer to the values very close to 1 as being *equal* to 1, since that is the case when working with limited-precision floating point numbers.



(a) MNIST (b) Fashion-MNIST (c) CIFAR

Fig. 3: Comparison of the angle between fake and regular gradients ($\theta(F, R)$) with the angle between two subsets of regular gradients ($\theta(R_1, R_2)$), averaged over 5 runs during honest training. The x-axis denotes the number of fake batches sent, also standing for the passage of training time during the first epoch.



(a) MNIST (b) Fashion-MNIST (c) CIFAR

Fig. 4: Comparison of the average magnitude values ($d(F, R)$ and $d(R_1, R_2)$) for fake and regular gradients, averaged over 5 runs during honest training. The x-axis denotes the number of fake batches sent, also standing for the passage of training time during the first epoch.

protocol might need to be extended beyond the first epoch for more complex tasks.

C. Detecting FSHA

With the claims validated, the questions of actual effectiveness remains: how well does SplitGuard defend against FSHA?

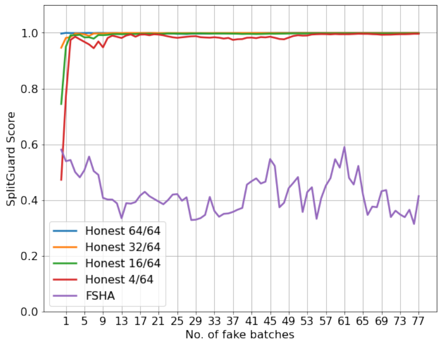
To show that SplitGuard can effectively detect a SplitNN server launching FSHA, we ran the attack for each of our datasets. Figure 5 displays the SplitGuard scores obtained during the first epoch of training by the clients against an honest server and a FSHA attacker, averaged over 5 runs. The P_F value is set to 0.1, and the B_F value varies.[5] We experimentally set the α and β values to 5 and 2 respectively, representing reasonable starting points, although we do not claim that they are optimal values.

The results displayed in Figure 5 indicate that the SplitGuard scores are distinguishable enough to enable detection by the client. The SplitGuard scores obtained with an honest server are very close or equal to 1, while the scores obtained against a FSHA server do not surpass 0.6. Notice that higher B_F values are expectedly more effective. For example, it takes slightly more time for the scores to get fixed around 1 for Fashion-MNIST with a B_F of $4/64$ compared to a B_F of 1. The same can be said for CIFAR10 as well, although it is evident that the B_F value should be set higher.

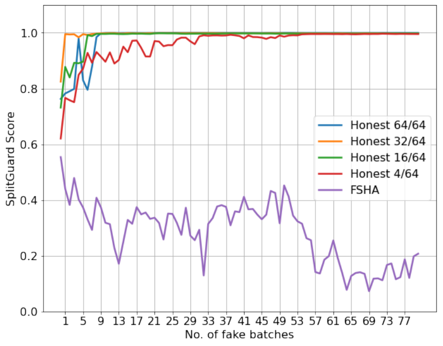
To assess more rigorously how accurate SplitGuard is at detecting FSHA, and likewise not reporting an attack during honest training, we define three candidate decision-making policies with different goals and test each one's effectiveness. A policy takes as input the list of SplitGuard scores obtained up to that point, and decides if the server is launching a training-hijacking attack or not. We set a threshold of 0.9 for these example policies. While the clients can choose different thresholds (Section VI-B), the results in 5 indicate that 0.9 is a sensible starting point. The three policies, also displayed in Algorithm 2 are defined as follows:

- *Fast:* Fix an early batch index. Report attack if the last score obtained is less than 0.9 after that index. The goal of this policy is to detect an attack as fast as possible, without worrying too much about a high false positive rate.

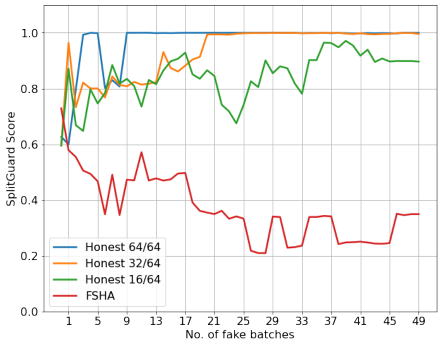
[5]Note that the B_F value does not affect the SplitGuard scores obtained against a FSHA server, since the client's loss function L_f is independent of the labels.



(a) MNIST



(b) Fashion-MNIST



(c) CIFAR

Fig. 5: SplitGuard scores obtained while training with an honest server, and a FSHA attacker during the first epoch of training, averaged over 5 runs. The x-axis displays the number of fake batches sent. The P_F value is set to 0.1, and the B_F values varies when training with an honest server.

Policy	**MNIST**			**F-MNIST**			**CIFAR10**		
	TP	FP	i	TP	FP	i	TP	FP	i
Fast	1	0.01	15	1	0.09	15	1	0.20	88
Avg-10	1	0	130	1	0.03	130	1	0.29	160
Avg-20	1	0	230	1	0.01	230	1	0.21	260
Avg-50	1	0	530	1	0	530	1	0.13	560
Voting	1	0	520	1	0	520	1	0.02	550

TABLE III: Attack detection statistics for the five example policies, collected over 100 runs of the first epoch of training with a FSHA attacker and an honest server. The true positive rate (TP) corresponds to the rate at which SplitGuard succeeds in detecting FSHA. The false positive rate (FP) corresponds to the share of honest training runs in which SplitGuard mistakenly reports an attack. The i field denotes the average batch index at which SplitGuard successfully detects FSHA.

Algorithm 2: Example Detection Policies

```
Function FAST(S: scores):
    return $S[-1] < 0.9$
Function AVG-K(S: scores, k: no. of scores to average):
    return mean$(S[-k:]) < 0.9$
Function VOTING(S: scores, c: group count, n: group size):
    votes = 0
    // default $c = 10$ and $n = 5$
    for $i$ from 0 to $c$ do
        group = S$[i \cdot n : (i+1) \cdot n]$
        if mean(group) $< 0.9$ then
            votes += 1
    return votes $> c/2$
```

- *Avg-k:* Report attack if the average of the last k scores is less than 0.9. This policy represents a middle point between the *Fast* and the *Voting* policies.
- *Voting:* Wait until a certain number of scores is obtained. Then divide the scores up to a fixed number of groups, calculate each group's average, and report attack if the majority of the mean values is less than 0.9. This policy aims for a high overall success rate (i.e. high true positive and low false positive rates). It can tolerate making decisions relatively later.

Note that these policies are not conclusive, and are provided as basic examples. More complex policies can be implemented to suit different settings. We will discuss the clients' decision-making process in more detail in Section VI-B.

Table III displays the detection statistics for each of these strategies obtained over 100 runs of the first epoch of training with a FSHA attacker and an honest server with a B_F of 1 and P_F of 0.1. For the *Avg-k* policy, we use k values of 10, 20, and 50, corresponding to roughly 100, 200, and 500 batches with a P_F of 0.1; this ensures that the policy can run within the first training epoch.[6] For the *Voting* policy, we set the group size to 5 and the group count to 10, again corresponding to around 500 batches with P_F 0.1. Finally, we set N, the index at which SplitGuard starts running, as 20 for MNIST and F-MNIST, and 50 for CIFAR10.

A significant result is that all the strategies achieve a perfect true positive rate (i.e. successfully detect all runs of FSHA). Expectedly, the *Fast* strategy achieves the fastest detection times as denoted by the i values in Table III, detecting in less than a hundred training batches all instances of the attack.

Another important observation is that the false positive rates increase as the model's performance decreases, moving from MNIST to F-MNIST and then CIFAR10. This means that more training time should be taken to achieve higher success rates in more complex tasks. This is not a troubling scenario, since as we will shortly observe the model not having a high performance also implies that the attack will be less effective. Nevertheless, the *Voting* policy achieves a false positive rate of

[6]With a batch size of 64, one epoch is equal to 938 batches for MNIST and F-MNIST, and 782 for CIFAR10.

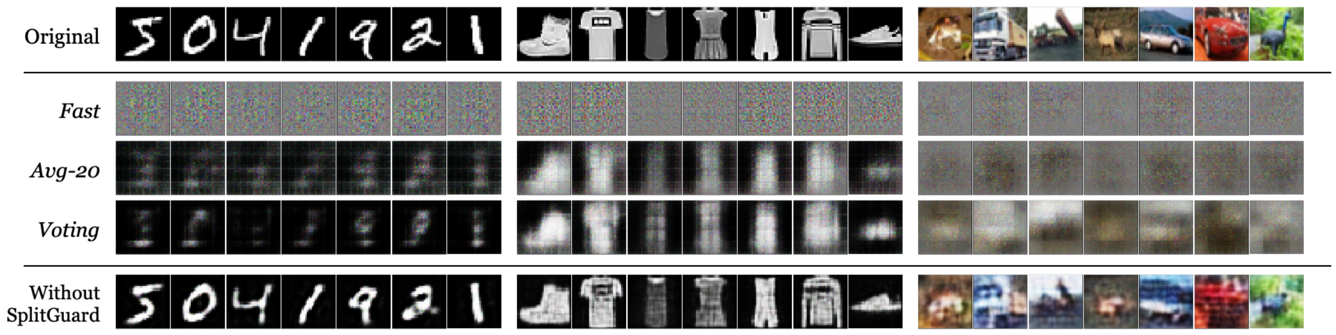


Fig. 6: Results obtained by a FSHA attacker for the MNIST, F-MNIST, and CIFAR10 datasets until the average detection times of the given policies as displayed in Table III. The first row displays the original images, and the last row displays the results obtained by a FSHA attacker able to run for an arbitrary duration without being detected.

0 for (F-)MNIST and 0.02 for CIFAR, indicating that despite the relatively high false positive rates of the *Avg-k* policies, better detection performance in less time is achievable through smarter policies.

D. What Does the Attacker Obtain Until Detection?

We now analyze what a FSHA adversary can obtain until the detection batch indices displayed in Table III. Figure 6 displays the results obtained by the attacker after the batch indices corresponding to the detection times of the given policies. Note that all these batch indices fall within the first training epoch.

It is visible that for the *Fast* policy, the attacker obtains not much more than random noise. This means that if a high false positive rate can be tolerated (e.g. privacy of the data is highly critical, and the server is distrusted), this policy can be applied to prevent any data leakage.

Unsurprisingly, the attack results get more accurate as the attacker is given more time. Nevertheless, especially for the more complex CIFAR10 task, the results obtained by the attacker against the *Voting* policy do not contain the distinguishing features of the original images. This highlights the effectiveness of the *Voting* policy, preventing significant information leakage with a relatively low false positive rate of 0.02. We would like to note once again that the policies described above are rather simplistic, and do not use the clients' full power, as will be discussed in Section VI-B.

Finally, the CIFAR10 results also give credibility to our previous statement that giving more time to the attacker for a more complex task should not be a cause of worry. After the same number of batches, the attacker's results for MNIST and Fashion-MNIST are more accurate than the CIFAR10 results.

VI. Discussion

In this section, we answer the following questions:

- What is the computational complexity of running SplitGuard (for clients)?
- How can the clients make a decision on whether the server is honest or not?
- Can the attacker detect SplitGuard? What happens if it can?
- What effect do the parameters have on the system?
- Can SplitGuard generalize to different scenarios?
- What are some concrete use cases for SplitGuard?

A. Computational Complexity

We now argue that SplitGuard does not incur a significant computational cost regarding time or space. Since SplitNN clients are already assumed to be able to run back-propagation on a few DNN layers, calculating the S value described in Equation 6 is a simple task.

Space-wise, although it might seem like storing the gradient vectors for potentially multiple epochs requires a significant amount of space, the clients in fact do not have to store all the gradient vectors. For each of the sets F, R_1, R_2, the clients have to maintain two quantities: a sum of all vectors in the set, and the average magnitude of the vectors in the set; the first has the dimensions of a single gradient vector, and the second is a scalar. More importantly, both of these quantities can be maintained in a running manner. This keeps the total space required by SplitGuard to $O(1)$ with respect to training time, equivalent to the space needed for three scalar values and three gradient vectors. For reference, the space required to store a single gradient vector in our experiments was 2.304 KB. Since the space requirement is independent of the total number of batches, it is possible to run SplitGuard for arbitrarily long training processes.

B. Clients' Decision-Making Process

We have described some makeshift decision-making policies in Section V-C, and in this subsection we discuss the clients' decision-making process in more depth, without focusing on a specific policy.

After each fake batch, clients can make a decision on whether the server is launching an attack or not. The main decision procedure is as follows:

1) Is the SG value **high or low**?
   a) If **high**, there are no problems. Keep training.
   b) If **low**, there are two possible explanations:
      i) The model has not learned enough yet. Keep going, potentially making changes.

ii) The server is launching an attack. Halt training.

Going back to the policies we have described in Section V-C, it can be seen that they did not consider the first explanation (1.b.i) of low scores, namely the model not having learned enough. As we will see, taking that into consideration could help reduce the false positive rates.

The outline contains two branching points: separating high and low scores, and explaining low scores.

Separating High and Low Scores. The process of separating high and low SplitGuard scores consists of two steps: setting the hyperparameters of the squashing function, and deciding on a threshold value in the interval $(0, 1)$. We consider two scenarios: the clients know or do not know the server model architecture.

If the clients know the architecture, then the clients can train the entire model using all or part of their local data, and gain a prior understanding of what S values (Equation 6) values to expect during honest training. The parameters α and β can then be adjusted to map these values very close to 1. In this scenario, since the clients' confidence on the accuracy of the method is expected to be higher, a relatively high threshold can be set, such as 0.95.

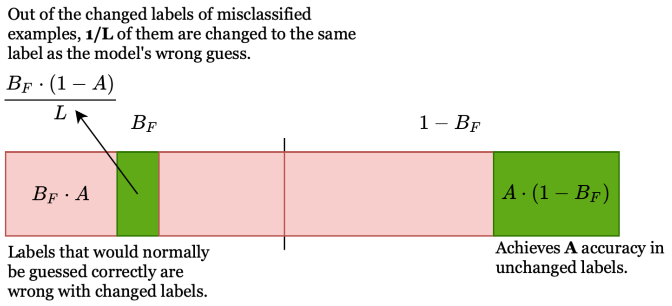
If the clients do not know the model architecture, then they should set the parameters α and β manually. Nevertheless, S values all lying within the interval $[-\pi, \pi]$ makes the clients' job easier. It is unreasonable to set extremely high α or β values since they will cause the squashing function to make sudden jumps, or map no value close to one. As our experiments also demonstrate, smaller values such as 5 and 2 are reasonable starting points.

Finally, note that the clients do not have to decide based on a single SplitGuard score. They can consider the entire history of the score, as depicted in Figure 5 and done in the *Avg-k* and *Voting* policies. For example, the score making a sudden jump to 1 and shortly going down to 0.5 does not imply honest training; similarly, the score making a sudden jump down to 0.5 after consistently remaining close to 1 does not strictly imply training-hijacking.

Explaining Low Scores. When a client decides that the SplitGuard score is low, it should choose between two possible explanations: either the model has not learned enough yet, or the server is launching a training-hijacking attack.

Informally, a low score indicates that fake gradients are not that different from regular gradients; the model behaves similarly when given fake batches and regular batches. In the domain of classification, *behaving similarly* is equivalent to having a similar classification accuracy. Then, the explanation that the model has not learned enough yet is more likely if the expected classification accuracy for a fake batch is close to the actual (expected) prediction accuracy. If these values are different but the SplitGuard score is still low, then the server is very likely launching an attack.

We can formulate the expected accuracy for a fake batch. Say the total number of labels is $L \in \mathbb{N}$ $(L \geq 2)$ and the overall model has classification accuracy $A \in [1/L, 1]$. Then



Out of the changed labels of misclassified examples, **1/L** of them are changed to the same label as the model's wrong guess.
$\frac{B_F \cdot (1-A)}{L}$
B_F
$1 - B_F$
$B_F \cdot A$
$A \cdot (1 - B_F)$
Labels that would normally be guessed correctly are wrong with changed labels.
Achieves **A** accuracy in unchanged labels.

Fig. 7: Expected classification accuracy in a fake batch with the share B_F of the labels randomized. The model normally has classification accuracy A with L labels.

the expected classification accuracy for a fake batch with the share $B_F \in [0, 1]$ of the labels randomized is

$$A_F = A \cdot (1 - B_F) + \frac{B_F \cdot (1 - A)}{L}. \quad (7)$$

Figure 7 explains this equation visually.

If the model terminates on the client-side (as in Figure 1b), then the clients already know the exact accuracy value. If that is not the case but the clients know the model architecture on the server side, then they can train the model using their local data, and obtain an estimate of the expected classification accuracy of the actual model during the first epoch. If even that is not possible, then in the worst-case the clients can train a linear classifier appended to their model to obtain a lower bound on the original model accuracy.[7]

Formalizing this discussion, for SplitGuard to be effective, it must be the case that $A >> A_F$. If $A_F \approx A$, then the clients' choice of B_F is not right, and they should increase it. Note that A_F is a linear function of B_F with the coefficient

$$-A + \frac{1}{L} - \frac{A}{L}.$$

Since $A \in [1/L, 1]$,

$$-A + \frac{1}{L} \leq 0$$

and

$$-A + \frac{1}{L} - \frac{A}{L} \leq 0$$

as well. Thus, A_F is indeed a monotonic function of B_F, and increasing B_F either keeps A_F constant or decreases it. Then when the clients decide that the SG value is low and that $A_F \approx A$, the best course of action is to increase B_F. If B_F is already 1, then clients should wait until the model becomes sufficiently accurate so that a completely randomized batch makes a difference. Note that as discussed previously, this is not a worrisome scenario, since the attack's effectiveness also relies on the model's adeptness.

[7]A related, interesting study concludes that what a neural network learns during its initial epoch of training can be *explained* by a linear classifier [23], in the sense that if we know the linear model's output, then knowing the main model's output provides almost no benefit in predicting the label. Note however that this does not hold for *any* linear classifier, but the optimal one.

Algorithm 3: Clients' decision-making process

```
$A$: Model's classification accuracy
$A_F$: Expected classification accuracy for a fake batch
$B_F$: Share of randomized labels in a fake batch
$N$: Number of initial batches to ignore
Function MAKE_DECISION($F, R$):
    if scores are high then
        Keep training.
    else if $A \approx A_F$ then
        if $B_F = 1$ then
            It is too early to detect. Wait.
        else
            Increase $B_F$.
        [Optional] Increase $N$.
    else
        The server is launching an attack. Stop training.
```

Finally, an alternative course of action is to increase N, discarding the initial group of gradients. Since the models behave randomly in the beginning, increasing N decreases the noise, and can help distinguish an honest server from a malicious one. Also note that increasing N is a reversible process, provided that clients store the gradient values.

With these discussions, we can finalize the clients' decision-making process as the function MAKE_DECISION, displayed in Algorithm 3.

C. Detection by the Attacker

An attacker can in turn try to detect that a client is running SplitGuard. It can then try to circumvent SplitGuard by using a legitimate surrogate model as described before.

If the server controls the model's output (Figure 1a), then it can detect if the classification error of a batch is significantly higher than the other ones. Since SplitGuard is a potential, though not the only, explanation of such behavior, it presents an opportunity for an attacker to detect it. However, the model behaving significantly differently for fake and regular batches also implies that the model is at a stage at which SplitGuard is effective. This leads to an interesting scenario: since the attack's and SplitGuard's effectiveness both depends on the model learning enough it seems as if the attack cannot be detected without the attacker detecting SplitGuard and vice versa.

We argue that this is not the case, due to the clients being in charge of setting the B_F value. For example, with the MNIST dataset for which the model obtains a classification accuracy around 98% after the first epoch of training, a B_F value of $4/64$ results in an expected classification accuracy of 91.8% for fake batches (Equation 7). The SplitGuard scores on the other hand displayed in Figure 5 being very close to one implies that an attack can be detected with such a B_F value. Thus, clients can make it difficult for an attacker to detect SplitGuard by setting the B_F value more smartly, rather than setting it blindly as 1 for better effectiveness.

Finally, we strongly recommend once again that a secure SplitNN setup follow the three-part setup shown in Figure 1b to prevent the clients sharing their labels with the server. This way, an attacker would not be able to see the accuracy of the model, and it would become significantly harder for it to detect SplitGuard.

D. Choosing Parameter Values

We have touched upon how the clients can decide on B_F, α, and β values, but we need to clarify the effects of the parameter values (mainly B_F, P_F, and N) for completeness. Each parameter involves a different trade-off:

- **Probability of sending a fake batch** (P_F).
  - $(+)$ Higher P_F values mean more fake batches, and thus a more representative sample of fake gradient values, increasing the effectiveness of the method.
  - $(-)$ Higher P_F values can also degrade model performance, since the server model will be learning random labels for a higher number of examples, and a higher share of the potentially scarce dataset will be allocated for SplitGuard.
- **Number of randomized labels in each batch** (B_F).
  - $(+)$ More random labels in a batch means that fake batches and regular batches behave even more differently, and the method becomes more effective.
  - $(-)$ Depending on the model's training performance, batches with entirely random labels can be detected by the server. One way to overcome this difficulty is to perform the loss computation on the client side.
- **Number of initial batches to ignore** (N).
  - $(+)$ A smaller N value means that the server's malicious behavior can be detected earlier, giving it less time to attack.
  - $(-)$ Since a model behaves randomly in the beginning of the training, the initial batches are of little value for our purposes. Computing SG scores for later batches will make it easier to distinguish honest behavior, but in return give the attacker more time.

E. Generalizing SplitGuard

In the form we have discussed so far, a question might arise regarding SplitGuard's effectiveness in different scenarios. We argue however that since the claims underlying SplitGuard are applicable to any kind of neural network learning on any kind of data, SplitGuard is generalizable to different data modalities, or more complex architectures. The only caveat, as discussed earlier, is that learning on a more complex dataset or with a more complex architecture would require more time for SplitGuard to be effective.

Another direction of generalization is towards different attacks. Although there are no training-hijacking attacks other than FSHA against which we can test SplitGuard, we claim that SplitGuard can generalize to future attacks as well. After all, SplitGuard relies only on the assumption that randomizing the labels affects an honest model more than it affects a malicious model. Thus, to go undetected by SplitGuard, an attack should either involve learning significant information about the original task, which would likely reduce the attack's effectiveness, or craft a different loss function for each label, which could easily be prevented by not sharing the labels with the server (Figure 1b).

Finally, SplitGuard also generalizes to multiple-client SplitNN settings. Each client can independently run SplitGuard, with their own choices of parameters. Each client would then be making a decision regarding its own training process. Alternatively, if the clients trust each other, they can choose one client to run SplitGuard in order to minimize its effect on performance loss, or they can combine their collected gradient values and reach a collective decision.[8] The latter scenario would be equivalent to a single client training with the aggregated data of all the clients.

F. Use Cases

We now describe three potential real-world use cases for SplitGuard, modeling clients with different capabilities at each scenario.

Powerful Clients. A group of healthcare providers decide to train a DNN using their aggregate data while maintaining data privacy. They decide on a training setup, and establish a central server.[9] Each client knows the model architecture and the hyperparameters, and preferably has access to the model's output as well (no label-sharing). The clients can train models using their local data to determine the parameters α and β. Each client can then run SplitGuard during their training turns and see if they are being attacked. This is an example scenario with the clients as powerful as possible, and thus represents the optimal scenario for running SplitGuard.

Intermediate Clients. The SplitNN server is a researcher, attempting to perform privacy-preserving machine learning on some private dataset of some data-holder (the client). The researcher designs the training procedure, but the data-holder actively takes part in the protocol. The data-holder thus has tight control over how its data is organized. The client cannot train a local model since it does not know the entire architecture, and should set the parameters α and β manually. Nevertheless, it can easily run SplitGuard by modifying the training data being used in the protocol.

Weak Clients. An application developer is the SplitNN server, and the users' mobile devices are the clients with private data. The clients do not know the model architecture, and cannot manipulate how their data is shared with the server. The application developer is in control of the entire process from design to execution. In this scenario, SplitGuard should be implemented at a lower-level, such as the ML libraries the mobile OS supports. However, even in that scenario, the application developer can implement a machine learning pipeline from scratch, without relying on any libraries. This is not an optimal scenario for running SplitGuard. There would have to be strict regulations, as well as gatekeeping by the OS provider (e.g. mandating that machine learning code must use one of the specified libraries) before SplitGuard could effectively be implemented for such clients.

[8] This is similar to the *Voting* policy described earlier, where the separation of scores into groups follows naturally from their distributions among the clients.

[9] Alternatively, members of the group can take turns acting as the SplitNN server in a P2P manner.

VII. Future Directions

We outline three possible avenues of future work related to SplitGuard: providing practical implementations, improving its robustness against detection by the attacker, and developing potentially undetectable attacks.

Although we provide a proof-of-concept implementation of SplitGuard, it should be readily supported by privacy-preserving machine learning libraries, such as PySyft [24]. That way, SplitGuard can be seamlessly integrated into the client-side processes of split learning pipelines.

As we have explained in Section VI-C, SplitGuard can potentially, although unlikely, be detected by the attacker, who can then start sending fake gradients from its legitimate surrogate model and regular gradients from its malicious model. This could again cause a significant difference between the fake and regular gradients, and result in a high SplitGuard score. However, a potential weakness of this approach by the attacker is that now the fake gradients result from two different models with different objectives. Suppose the attacker detects SplitGuard at the 200th batch, and starts using its legitimate model. Then the fake gradients within the first 200 batches will be computed using a malicious model, and those after the 200th batch will be computed using the legitimate model. Clients can potentially detect this switch in models, and gain the upper hand. This is another point for which future improvement might be possible.

Finally, turning the tables, it might be possible to modify the existing attacks, or propose novel attacks to produce high SplitGuard scores, very likely at the cost of effectiveness. This represents another line of future work concerning SplitGuard.

VIII. Conclusion

In this paper, we presented SplitGuard, a method for SplitNN clients to detect if they are being targeted by a training-hijacking attack [11] or not. We described the theoretical foundations underlying SplitGuard, experimentally evaluated its effectiveness, and discussed at depth many issues related to its use. We conclude that when used appropriately, and in a secure setting without label-sharing, a client running SplitGuard can successfully detect training-hijacking attacks and leave the attacker empty-handed.

References

[1] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," *arXiv:2005.14165 [cs]*, July 2020. arXiv: 2005.14165.

[2] A. Halevy, P. Norvig, and F. Pereira, “The Unreasonable Effectiveness of Data,” *IEEE Intelligent Systems*, vol. 24, pp. 8–12, Mar. 2009.

[3] N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso, “The computational limits of deep learning,” *arXiv preprint arXiv:2007.05558*, 2020.

[4] G. J. Annas, “HIPAA Regulations — A New Era of Medical-Record Privacy?,” *New England Journal of Medicine*, vol. 348, pp. 1486–1490, Apr. 2003.

[5] R. T. Mercuri, “The HIPAA-potamus in health care data security,” *Communications of the ACM*, vol. 47, pp. 25–28, July 2004.

[6] P. Vepakomma, O. Gupta, T. Swedish, and R. Raskar, “Split learning for health: Distributed deep learning without sharing raw patient data,” *arXiv preprint arXiv:1812.00564*, 2018.

[7] O. Gupta and R. Raskar, “Distributed learning of deep neural network over multiple agents,” *arXiv:1810.06060 [cs, stat]*, Oct. 2018. arXiv: 1810.06060.

[8] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečný, S. Mazzocchi, H. B. McMahan, T. Van Overveldt, D. Petrou, D. Ramage, and J. Roselander, “Towards Federated Learning at Scale: System Design,” *arXiv:1902.01046 [cs, stat]*, Mar. 2019. arXiv: 1902.01046.

[9] J. Konečný, H. B. McMahan, D. Ramage, and P. Richtárik, “Federated Optimization: Distributed Machine Learning for On-Device Intelligence,” *arXiv:1610.02527 [cs]*, Oct. 2016. arXiv: 1610.02527.

[10] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, “Federated Learning: Strategies for Improving Communication Efficiency,” *arXiv:1610.05492 [cs]*, Oct. 2017. arXiv: 1610.05492.

[11] D. Pasquini, G. Ateniese, and M. Bernaschi, “Unleashing the Tiger: Inference Attacks on Split Learning,” *arXiv:2012.02670 [cs]*, Jan. 2021.

[12] Y. LeCun, C. Cortes, and C. Burges, “Mnist handwritten digit database,” *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, vol. 2, 2010.

[13] H. Xiao, K. Rasul, and R. Vollgraf, “Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms,” *arXiv preprint arXiv:1708.07747*, 2017.

[14] A. Krizhevsky, G. Hinton, *et al.*, “Learning multiple layers of features from tiny images,” 2009.

[15] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[16] S. Ruder, “An overview of gradient descent optimization algorithms,” *arXiv:1609.04747 [cs]*, June 2017. arXiv: 1609.04747.

[17] P. Vepakomma, T. Swedish, R. Raskar, O. Gupta, and A. Dubey, “No peek: A survey of private distributed deep learning,” *arXiv preprint arXiv:1812.03288*, 2018.

[18] A. Singh, P. Vepakomma, O. Gupta, and R. Raskar, “Detailed comparison of communication efficiency of split learning and federated learning,” *arXiv preprint arXiv:1909.09145*, 2019.

[19] E. Erdogan, A. Kupcu, and A. E. Cicek, “Unsplit: Data-oblivious model inversion, model stealing, and label inference attacks against split learning,” *arXiv preprint arXiv:2108.09033*, 2021.

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

[21] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” *arXiv:1412.6980 [cs]*, Jan. 2017. arXiv: 1412.6980.

[22] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[23] D. Kalimeris, G. Kaplun, P. Nakkiran, B. Edelman, T. Yang, B. Barak, and H. Zhang, “Sgd on neural networks learns functions of increasing complexity,” *Advances in Neural Information Processing Systems*, vol. 32, pp. 3496–3506, 2019.

[24] T. Ryffel, A. Trask, M. Dahl, B. Wagner, J. Mancuso, D. Rueckert, and J. Passerat-Palmbach, “A generic framework for privacy preserving deep learning,” *arXiv preprint arXiv:1811.04017*, 2018.