# VERIDL: Verifiable Federated Deep Learning

## ABSTRACT

Federated learning is an emerging technique that enables multiple participants to collaboratively learn a centralized global model from the distributed training data. The participants compute the local model updates from their individual data. These local updates are sent to the server for updating the global model. Although Federated learning enables privacy-preserving learning, it raises the serious integrity concern of how to verify the correctness of local model updates by potentially untrusted participants in an adversarial environment. In this paper, we design VeriDL, a framework that can authenticate the correctness of local model updates with small overhead. By adapting a cryptographic tool named bilinear mapping to Federated learning, VeriDL enables the server to verify the correctness of local model updates without access to the local data of the individual participants, but instead through a cryptographic proof. Our experiments demonstrate the efficiency and effectiveness of VeriDL.

**ACM Reference Format:**

. 2019. VERIDL: Verifiable Federated Deep Learning. In *KDD'19: 25TH ACM SIGKDD Conference ON Knowledge Discovery and Data Mining,August 4 - 8, 2019, Anchorage, Alaska USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

The recent abrupt advances in deep learning (DL) [2, 13] have led to breakthroughs in many different fields. Most of the DL approaches require centralizing the training data to learn a global model. However, centralized collection of photos, speech, video, and other information (e.g., health data) from millions of individuals raises serious privacy risks. As the data scale grows significantly and models become much more complex, training models increasingly require distributing the learning process over multiple machines or devices [12]. *Federated learning*, termed by Google [12], emerged as an attractive framework that allows multiple participants to learn a global model on their own inputs without sharing their private data. At high level, the Federated learning model works as following - The server initializes a deep neural network (DNN) as the global model. At each round, the server selects a random subset of participants to download the latest global model and compute an update to the global model with their local data. The process repeats until the global model reaches convergence. Google has been intensively testing this new paradigm in its recent projects such as Gboard, the Google Keyboard [14], on Android.

Although Federated learning enables privacy-preserving learning with high accuracy, it is vulnerable to the presence of adversary and system failures. For example, an attacker can control one or several participants to "backdoor" the system so that the global model behaves incorrectly on attacker-chosen inputs [1]. Or the system suffers from Byzantine failures since some participants may fail to generate correct local model updates due to software bugs, computation errors and network crashes [3, 4].

The quality of the global model is sensitive to the local model updates; a small amount of error on the local model updates may easily prevent the convergence of the learning algorithm [3, 25]. To ensure the accuracy of the global model, only the correct local model updates should be aggregated into the global model. Therefore, it raises a serious concern of the *integrity* (i.e., correctness) of local model updates. However, verifying the correctness of local model updates is not straightforward, as the server cannot access the local data. A possible solution is that the server performs privacy-preserving deep learning to re-compute the local model updates on encrypted input by using homomorphic encryption (HE) [8, 10], and compares the results with what are sent by the participants. Though correct, this approach may incur expensive overhead due to the high complexity of HE. Furthermore, since HE only can support low degree polynomials, most of the activation functions that are commonly used in DNNs (e.g., ReLU, Sigmoid, and Tanh) have to be approximated. By using the approximated activation function, the server cannot compute the exact local model updates and thus compare them with the participants' uploaded values for correctness verification. Some existing DL verification methods (e.g., [7, 9]) only can provide a probabilistic integrity guarantee. Our goal is to provide a deterministic integrity guarantee (i.e., with 100%

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

| Method | Integrity guarantee | Learning accuracy | Activation function |
|---|---|---|---|
| Homomorphic encryption [8, 10] | Deterministic | Lossy | Restricted |
| SafetyNets [7] | Probabilistic | Lossy | Restricted |
| VeriDeep [9] | Probabilistic | Lossy | Broad |
| VeriDL **(ours)** | Deterministic | Lossless | Broad |

**Table 1: Comparison with Existing Work**

certainty). Furthermore, most of the existing DL verification methods (e.g., [7, 9]) have a few restrictions on the activation function (must be polynomials with integer coefficients) and weights/inputs (must be integers), which limit their use in Federated learning.

**Our contributions.** We design VeriDL, a framework that supports efficient integrity verification of local model updates by adversarial participants. VeriDL provides a deterministic integrity guarantee of the local model updates, without any constraint on the activation function, weights, and data types of inputs of the DNN model. Table 1 shows a brief comparison between VeriDL and the most relevant existing methods. The key idea of VeriDL is that each participant constructs a cryptographic proof of the local model updates, and sends both updates and the proof to the server. The proof enables the server to authenticate the correctness of the local model updates *without access to the local data*. Here the correctness means the *authenticity* (i.e., the local model updates are indeed computed faithfully from the local data) and *convergence* (i.e., they are not the intermediate results of the learning process at the participant side).

We make the following contributions. First, we design an efficient proof construction procedure by adapting *bilinear mapping*, a pairing-based cryptographic tool, to Federated learning. The adaption is not easy, as bilinear mapping cannot deal with decimal and negative values. Thus we extend the bilinear mapping method significantly to handle decimal and negative values in DNN. Second, we design a lightweight verification method that can authenticate the correctness of local model updates without access to the original data, but through the proof instead. We provide formal security proof of VeriDL. Finally, we implement VeriDL prototype, deploy it on a DL system, and evaluate its performance. Our experimental results demonstrate the efficiency and effectiveness of VeriDL. The verification by VeriDL is faster than privacy-preserving deep learning (on encrypted data) [8, 10] by more than three orders of magnitude.

The paper is organized as following. Section 2 introduces the preliminaries. Section 3 presents the basic operations of DNN, and Section 4 presents the details of how to authenticate the outputs of these basic operations. Section 5 discusses our experimental results. Section 6 compares our work with the related work. Section 7 concludes the paper.

| Notation | Meaning |
|---|---|
| $m$ | # of features |
| $N$ | # of input samples |
| $L$ | # of hidden layers |
| $(\vec{x}, y)$ | An input sample ($\vec{x}$: features; $y$: label) |
| $n_k^\ell$ | The $k$-th neuron on the $\ell$-th hidden layer |
| $z_k^\ell/a_k^\ell$ | The weighted sum/activation of $n_k^\ell$ |
| $w_{jk}^\ell$ | The weight between $n_j^{\ell-1}$ and $n_k^\ell$ |
| $\delta_k^\ell$ | The error signal on $n_k^\ell$ |
| $G$ | A multiplicative cyclic group for bilinear mapping |
| $g$ | The generator of the cyclic group $G$ |

**Table 2: Notation table**

## 2 PRELIMINARIES

In this section, we introduce the preliminaries. The notations used in the paper are shown in Table 2.

### 2.1 Bilinear Mapping

Our authentication protocol relies on a cryptographic protocol named *bilinear mapping*. Let $G$ and $G_T$ be two multiplicative cyclic groups of finite order $p$. Let $g$ be a generator of $G$. A bilinear group mapping $e$ is defined as $e : G \times G \rightarrow G_T$, which has the following properties: (1) *Bilinearity*: $e(g^a, g^b) = e(g, g)^{ab}$, where $a, b \in \mathbb{Z}_p$; (2) *Computability*: The bilinear map $e$ is efficiently computable by $G \times G$ for any pairs; (3) *Non-degeneracy*: $e(g, g) \neq 1$, i.e., all pairs of the source group do not map to the identity of the target group.

### 2.2 System Model

We consider the general Federated learning model [15] that consists of a parameter server, and multiple participants. Each participant has a set of private labeled data samples that he is willing to share with neither the server nor other participants. The server aims to train a global DNN from the data samples of all participants. We assume that all participants agree on a common DNN structure and a common learning objective. The core Federated learning protocol consists of two phases, namely *setup* and *learning*. During the *setup* phase, the server initializes a DNN as a global model. Then during the *learning* phase, the server randomly selects a subset of participants and sends them the latest DNN $M^t$ at each round $t$. Each selected participant $P_i$ computes independently the local model update $\Delta L_i$ (i.e., the accumulated update of model parameters when the local model reaches convergence) on his own data samples. Afterwards, these participants asynchronously upload the local model updates to the server. The server aggregates the local updates and updates the global DNN $M^t$ to be $M^{t+1}$. The learning process repeats until the global model reaches convergence.

## 2.3 Attack Model

We assume that the server computes the global model honestly. However, the participants are unreliable and unpredictable. For example, some participants may terminate their learning process early before it reaches the convergence locally due to software bugs or system crashes. Or a *lazy* participant sends arbitrary values to the server as the local model update, hoping to benefit from other participants with little cost from his side. Even worse, some participants are malicious and aim to pollute the global model by crafting the local model updates carefully [1].

In this paper, we only focus on the *integrity of model updates*. We do not consider *data integrity*. We assume the integrity of local data (including both features and labels) has been verified by the server (e.g., by using hash and signature techniques [5] that support data integrity verification without revealing real data values). Therefore, the local data cannot be tampered with by the attacker. The attacker also cannot insert adversarial examples into the local dataset [1].

## 2.4 Verification Goal

It has been shown that small amounts of error on the local model updates may easily prevent the convergence of the global model [3, 25]. Therefore, it is important that the server verifies the correctness of the received local model updates before aggregating them into the global model. In particular, given a participant $P$ and its uploaded local model update $\Delta L$, the server aims to verify the correctness of $\Delta L$, where correctness means the *authenticity* and *convergence* of $\Delta L$:

- *Authenticity of* $\Delta L$: whether $\Delta L$ has indeed been computed honestly on $P$'s local data;
- *Convergence of* $\Delta L$: whether $\Delta L$ reaches convergence in the local model.

The challenge is to enable the server to verify convergence without access to the local data of $P$.

## 2.5 Authentication Protocol

We adapt the definition of authentication protocols in [19] to our setting. Formally,

Definition 2.1 (Authentication protocol). *Let $W$ be the set of weight parameters in a DNN, and $D$ be a collection of data samples. Let $\Delta L$ be the parameter update after training the DNN on $D$. The authentication protocol is a collection of the following three polynomial-time algorithms,* **genkey** *for protocol setup,* **certify** *for verification preparation, and* **verify** *for verification.*

- $\{p_k\} \leftarrow \mathbf{genkey}()$*: It outputs a public key* $p_k$*;*
- $\{(\Delta L, \Pi\} \leftarrow \mathbf{certify}(\mathbf{D}, \mathbf{W}, \mathbf{p_k})$*: Given the data collection $D$, the DNN model parameters $W$, and a public key $p_k$, it returns the model update $\Delta L$, along with its proof $\Pi$;*
- $\{\text{accept}, \text{reject}\} \leftarrow \mathbf{verify}(\mathbf{W}, \mathbf{\Delta L}, \mathbf{\Pi}, \mathbf{p_k})$*: Given the DNN model parameters $W$, the local model update $\Delta L$, the proof $\Pi$, and the public key $p_k$, it outputs either* accept *or* reject.

## 2.6 Security Model

In this paper, we consider the malicious adversary who has full knowledge of the authentication protocol. Next, we define the security of the authentication protocol against such malicious adversary.

Definition 2.2 (Security). *Let* **Auth** *be an authentication scheme {***genkey***,* **certify***,* **verify***}. Let also* **Adv** *be a probabilistic polynomial-time adversary that is only given $p_k$. The adversary has unlimited access to all algorithms of* **Auth***. Then, for any neural network with parameters $W$ and any dataset $D$,* **Adv** *returns a wrong model update $\Delta L'$, and a proof $\Pi'$. The authentication scheme* **Auth** *is secure if for for all $p_k$ generated by the* **genkey** *scheme, and for any probabilistic polynomial-time adversary* **Adv***, it holds that*

$$Pr(\{\Delta L', \Pi'\} \leftarrow \mathbf{Adv}(D, W, p_k); accept \leftarrow \mathbf{verify}(W, \Delta L', \Pi', p_k)) = 0.$$

Intuitively, the authentication protocol is secure if with 100% certainty the incorrect local update cannot escape from verification.

## 3 BASIC DNN OPERATIONS

In this section, we present the basic operations of training a DNN. We will explain how to verify the output of these operations in Section 4.

A DNN consists of several layers, including the input layer (data samples), the output layer (the predicted labels of the data samples), and a number of hidden layers.

During the feedforward computation, for $n_k^\ell$, its weighted sum $z_k^\ell$ is defined as:

$$z_k^\ell = \begin{cases} \sum_{i=1}^m x_i w_{ik}^\ell & \text{if } \ell = 1 \\ \sum_{j=1}^{d_{\ell-1}} a_j^{\ell-1} w_{jk}^\ell & \text{otherwise,} \end{cases} \tag{1}$$

where $x_i$ is the $i$-th feature of the input $\vec{x}$, and $d_i$ is the number of neurons on the $i$-th hidden layer. The activation $a_k^\ell$ is calculated as follows:

$$a_k^\ell = \sigma(z_k^\ell), \tag{2}$$

where $\sigma$ is the activation function. We allow a broad class of activation functions such as sigmoid, ReLU (rectified linear unit), and hyperbolic tangent.

On the output layer, the output $o$ is generated by following the same way as the hidden layers:

$$o = \sigma(z^o) = \sigma(\sum_{j=1}^{d_L} a_j^L w_j^o), \tag{3}$$

where $w_j^o$ is the weight that connects $n_j^\ell$ to the output neuron.

In this paper, we mainly consider the mean square error (MSE) as the cost function. For any sample $(\vec{x}, y) \in D$, the cost $C(\vec{x}, y; W)$ is measured as the difference between the label $y$ and the output $o$:

$$C(\vec{x}, y; W) = C(o, y) = \frac{1}{2}(y - o)^2. \tag{4}$$

The error $E$ is calculated as the average error for all samples:

$$E = \frac{1}{N} \sum_{(\vec{x}, y) \in D} C(\vec{x}, y; W). \tag{5}$$

We note that some cost functions such as cross-entropy require to release the original labels. VeriDL does not support such cost functions due to the privacy concern.

In the backpropagation process, gradients are calculated to update the weights in the neural network. According to the chain rule of back propagation [13], for any sample $(\vec{x}, y)$, the error signal $\delta^o$ on the output neuron is

$$\delta^o = \nabla_o C(o, y) \odot \sigma'(z^o) = (o - y)\sigma'(z^o). \tag{6}$$

While the error signal $\delta^\ell_k$ at the $\ell$-th hidden layer is

$$\delta^\ell_k = \begin{cases} \sigma'(z^\ell_k) w^o_k \delta^o & \text{if } \ell = L, \\ \sigma'(z^\ell_k) \sum_{j=1}^{d_{\ell+1}} w^{\ell+1}_{kj} \delta^{\ell+1}_j & \text{otherwise.} \end{cases} \tag{7}$$

where $\ell = L$ indicates the last hidden layer.

The derivative for each weight $w^\ell_{jk}$ is computed as:

$$\frac{\partial C}{\partial w^\ell_{jk}} = \begin{cases} x_j \delta^\ell_k & \text{if } \ell = 1 \\ a^{\ell-1}_j \delta^\ell_k & \text{otherwise.} \end{cases} \tag{8}$$

Then the weight increment $\Delta w^\ell_{jk}$ is

$$\Delta w^\ell_{jk} = -\frac{\eta}{N} \sum_{(\vec{x}, y) \in D} \frac{\partial C}{\partial w^\ell_{jk}}, \tag{9}$$

where $\eta$ is the learning rate. Finally, the weight is updated as

$$w^\ell_{jk} = w^\ell_{jk} + \Delta w^\ell_{jk}. \tag{10}$$

After the weights are updated, the learning process runs the feedforward process followed by the backpropagation process until it reaches convergence (e.g., the error $E$ does not change any more).

## 4 AUTHENTICATION METHOD

In this section, we explain the details of our authentication protocol. The **genkey** protocol is straightforward. The server picks a pairing function $e$ on two sufficiently large cyclic groups $G$ and $G_T$ of order $p$. Also the server finds a generator $g \in G$. The server outputs the public key $p_k = \{g, G, G_T, e\}$. Therefore, in the following discussions, we only focus on the **certify** and **verify** protocols. For simplicity, we assume that both training and validation are conducted on all data

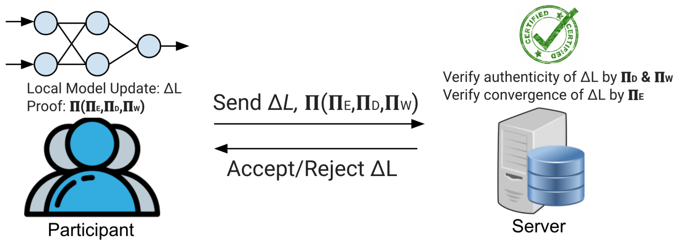


**Figure 1: Overview of our authentication method**

samples. Our authentication method is compatible with various optimization algorithms such as batch and stochastic gradient descent.

**Solution in a nutshell.** Consider a participant $P$ which has a local dataset $D$. Besides sending the local model updates $\Delta L$ to the server, $P$ also sends two errors $E_1$ and $E_2$, which are the errors on $D$ when the local model reaches convergence (as $P$ claimed) and by running an additional round of backpropagation and feedforward process after convergence, respectively. $P$ constructs a cryptographic proof of $E_1$ and $E_2$. The server can verify both authenticity and convergence of $\Delta L$ by the following two steps: (1) *Authenticity verification*: Without access to the local data of $P$, the server verifies if the error $E_1$ is correct through the proof; (2) *Convergence verification*: The server verifies if $E_2$ is correct. Then the server verifies if $E_1$ and $E_2$ satisfy the convergence condition. Figure 1 illustrates the authentication method.

### 4.1 Certify Protocol

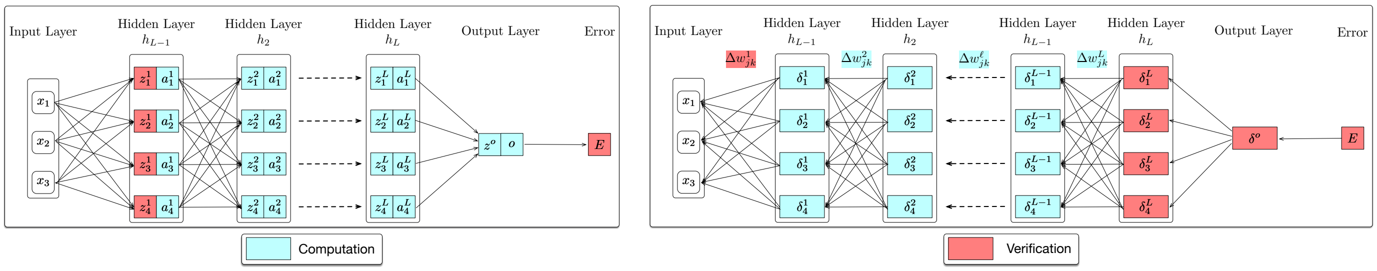
To enable the server to verify $E_1$ and $E_2$ without access to the private sample $(\vec{x}, y)$, our Certify protocol allows the participant to send the encrypted format of $(\vec{x}, y)$ to the server. These ciphertext values enable the server to calculate $E_1$ and $E_2$ without access to any local data point in $D$. In particular, the proof $\Pi$ is constructed as:

$$\Pi = \{\Pi_E, \Pi_D, \Pi_W\},$$

where $\Pi_E = \{E_1, E_2\}$, $\Pi_W = \{\Delta w^1_{jk}\}$, which is the increment to the weights between the input and first hidden layer incurred by the additional back-propagation, and

$$\Pi_D = \{\{g^{x_i}\}, g^y, \{z^1_k\}, \{\hat{z}^1_k\}, g^{\delta^o}, \{\delta^L_k\} | \forall (\vec{x}, y) \in D\},$$

where $z^1_k$ and $\hat{z}^1_k$ are the weighted sum of the $n^1_k$ (Formula 1) at the time of convergence and one additional round after convergence, and $\delta^o$ and $\{\delta^L_k\}$ are the error signals on the output and last hidden layer at convergence respectively. We must note that disclosing ($\{z^1_k\}$, $\{\hat{z}^1_k\}$, $\{\delta^L_k\}$, $\{\Delta w^1_{jk}\}$), which is just a small fraction of the intermediate results, reveals very limited information about the input local data. Therefore, it does not violate the privacy requirement.



(a) Feedfoward (b) Backpropagation

**Figure 2: An overview of the *verify* protocol**

## 4.2 Verify Protocol

The verification process consists of three steps: (1) one feedforward to verify the correctness of $E_1$; (2) one backpropagation to update weights; and (3) another feedforward to verify the correctness of $E_2$. Figure 2 illustrates the verification and computations needed in these three steps. Next, we discuss these steps in details.

### *4.2.1 Step 1. Verification of $E_1$.*

**Step 1.1 Verification of $\{z^1_k\}$.** Prior to verification, the server is aware of $w^1_{ik}$. The server also obtains $\{g^{x_i}\}$ and $\{z^1_k\}$ from the proof $\Pi$. To verify the correctness of $\{z^1_k\}$, for each $z^1_k$, the server checks if the following is true:

$$\Pi e(g^{x_i}, g^{w^1_{ik}}) \overset{?}{=} e(g,g)^{z^1_k}. \tag{11}$$

**Step 1.2 Calculation of $o$.** Once the server verifies the correctness of $\vec{z}^1$, it calculates the activation of the hidden layers and thus the output $o$ (Formula (2) and (3)).

**Step 1.3 Verification of $E_1$.** The server checks if the following is true:

$$\Pi_{(\vec{x},y)\in D} e(g^{y-o}, g^{y-o}) \overset{?}{=} e(g,g)^{2NE_1}, \tag{12}$$

where $g^{y-o} = g^y * g^{-o}$. Note that $g^y$ is included in the proof $\Pi_D$. The server computes $g^{-o}$ after learning $o$ from Step 1.2.

### *4.2.2 Step 2. Weight update.*

**Step 2.1 Verification of error signal on the output layer.** The server verifies the correctness of $g^{\delta^o}$ returned by the participant. Note that the participant cannot simply disclose $\delta^o$ to the server since the server already learns $\sigma'(z^o)$ and $o$. Following Formula (6), the server can easily infer the label $y$ if $\delta^o$ is provided. Therefore, $\Pi_D$ only includes $g^{\delta^o}$. In particular, the server verifies if

$$e(g^{-o}g^{y}, g^{-\sigma'(z^o)}) \overset{?}{=} e(g, g^{\delta^o}), \tag{13}$$

where $g^{-o}$ and $g^{-\sigma'(z^o)}$ are computed by the server, and $g^y$ and $g^{\delta^o}$ are from the proof.

**Step 2.2 Verification of error signals on the last hidden layer.** The server verifies the correctness of $\delta^L_k$ (Formula (7)), i.e., the error signal on the $k$-th neuron on the last hidden layer, by checking if

$$e(g^{w^o_k\sigma'(z^L_k)}, g^{\delta^o}) \overset{?}{=} e(g,g)^{\delta^L_k}, \tag{14}$$

where $g^{w^o_{kj}\sigma'(z^L_k)}$ is computed by the server, and $\delta^L_k$ and $g^{\delta^o}$ are obtained from the proof. Although the error signals on the last hidden layer, i.e., $\{\delta^L_k\}$, are presented as plaintext in the proof, it does not disclose $\delta^o$ to the server. Thus, it does not leak the input data.

**Step 2.3 Adjustment of weights between hidden layers.** The server calculates the error signal of other hidden layers by following Formula (7). Then with the knowledge of the activation on every hidden layer (learned by Step 1.2), the server computes the derivatives of the weights (Formula 8) on the hidden layers to update the weights between consecutive hidden layers (Formula 9 and 10).

**Step 2.4 Verification of weight increment between input and first hidden layer.** Without access to the input feature $x_j$, the server obtains $\Delta w^1_{jk}$ from the proof $\Pi$ and verifies its correctness by checking:

$$\Pi_{(\vec{x},y)\in D} e(g^{x_j}, g^{\eta\delta^1_k}) \overset{?}{=} e(g^{\Delta w^1_{jk}}, g^{-N}). \tag{15}$$

Note that $g^{x_j}$ and $\Delta w^1_{jk}$ are included in the proof, and $g^{\eta\delta^1_k}$ and $g^{-N}$ are calculated by the server. After $\Delta w^1_{jk}$ passes the verification in Formula (15), the server updates the weight by Formula (10).

*4.2.3 Step 3. Verification of $E_2$.* The server follows the same procedure of Step 1 to verify $E_2$, using the updated weights. We omit the detailed discussion.

## 4.3 Dealing with Decimal & Negative Values

A weakness of bilinear pairing is that it cannot use decimal and negative values as the exponent in $g^e$. Therefore, the verification in Formula (11 - 15) cannot be performed at ease. To address this problem, we extend the bilinear pairing protocol to handle decimal and negative values.

**Decimal values.** We design a simple method that conducts decimal arithmetic in an integer field without any accuracy loss. In general, consider checking if $b * c \stackrel{?}{=} e$, where $b$, $c$ and $e$ are three variables that may hold decimal values, let $L_D$ be the maximum number of bits after the decimal point allowed for any value. We define a new operator $f(\cdot)$ where $f(x) = x * 2^{L_D}$. Obviously, $f(x)$ must be an integer. We pick two cyclic groups $G$ and $G_T$ of sufficiently large order $p$ such that $f(x)f(y) \in Z_p$. Thus, we have $g^{f(x)} \in G$, and $e(g^{f(x)}, g^{f(y)}) \in G_T$. To make the verification in Formula (13) and (14) compatible with decimal values, we check if $e(g^{f(b)}, g^{f(c)}) \stackrel{?}{=} e(g,g)^{f(e)}$ instead. Obviously, if $e(g^{f(b)}, g^{f(c)}) = e(g,g)^{f(e)}$, it is natural that $b * c = e$. The verification in Formula (11), (12) and (15) is accomplished in the same way, except that the involved values should be raised by $2^{L_D}$.

**Negative values.** Formula (11 - 15) check for a given pair of vectors $\vec{u}, \vec{v}$ of the same size, whether $\sum u_i v_i = z$. Note that the verification in Formula (13) and (14) can be viewed as a special form in which both $\vec{u}$ and $\vec{v}$ only include a single scalar value. Also note that $u_i$, $v_i$ or $z$ may hold negative values. Before we present our methods to deal with negative values, we first define an operator $[\cdot]$ such that $[x] = x \bmod p$. Without loss of generality, we assume that for any $\sum u_i v_i = z$, $-p < u_i, v_i, z < p$. We have the following lemma.

Lemma 4.1. *For any pair of vectors $\vec{u}, \vec{v}$ of the same size, and $z = \sum u_i v_i$, we have*

$$[\sum [u_i][v_i]] = \begin{cases} z & if\, z \geq 0 \\ z + p & otherwise. \end{cases}$$

Due to the limited space, we leave the proof of Lemma 4.1 in the Appendix. Following Lemma 4.1, verifing if $\sum u_i v_i \stackrel{?}{=} z$ is equivalent to checking if

$$\Pi e(g^{[u_i]}, g^{[v_i]}) \stackrel{?}{=} \begin{cases} e(g,g)^z & \text{if } z \geq 0 \\ e(g,g)^{(z+p)} & \text{otherwise.} \end{cases} \quad (16)$$

Next, we focus on Formula (11) and discuss our method to handle negative values. First, from Lemma 4.1, we can see that for any $x_i$ and $w^1_{ik}$, if $x_i w^1_{ik} \geq 0$, then $[x_i][w^1_{ik}] = x_i w^1_{ik}$; otherwise, $[x_i][w^1_{ik}] = x_i w^1_{ik} + p$. Therefore, to prove $z^1_k = \sum x_i w^1_{ik}$, the participant includes a flag $sign_i$ for each $x_i$ in the proof, where

$$sign_i = \begin{cases} + & \text{if } x_i \geq 0 \\ - & \text{otherwise.} \end{cases}$$

Meanwhile, for each $z^1_k$, the participant prepares two values $p^1_k = \sum_{i:x_i w^1_{ik} \geq 0} x_i w^1_{ik}$ and $n^1_k = \sum_{i:x_i w^1_{ik} < 0} x_i w^1_{ik}$, and includes them in the proof.

In the verification phase, since the server is aware of $w^1_{ik}$, with the knowledge of $sign_i$ in the proof, it can tell if $x_i w^1_{ik} \geq 0$ or not. So the server first verifies if

$$\Pi_{i:x_i w^1_{ik} \geq 0} e(g^{[x_i]}, g^{[w^1_{ik}]}) \stackrel{?}{=} e(g,g)^{p^1_k},$$

and

$$\Pi_{i:x_i w^1_{ik} < 0} e(g^{[x_i]}, g^{[w^1_{ik}]}) \stackrel{?}{=} e(g,g)^{n^1_k + p},$$

where $g^{[x_i]}$ is included in the proof, and $g^{[w^1_{ik}]}$ is computed by the server. Next, the server checks if $p^1_k + n^1_k \stackrel{?}{=} z^1_k$.

We are aware that our method reveals the fact if the participant's input features are positive or negative. However, we argue that this piece of disclosed information has very limited impact on privacy as it cannot enable the adversary to infer the local data.

### 4.4 Security Analysis

In this section, we discuss the security of VeriDL. We have th following theorem.

Theorem 4.1. *Our authentication approach,* VeriDL, *is secure (Definition 2.2).*

**Proof Sketch.** Due to the space limit, we provide the proof sketch here. The formal proof can be found in the Appendix. Let $\Delta L'$ be the incorrect local model update provided by an adversary **Adv**. If **Adv** provides the correct proof $\Pi$, it must fail the verification in Formula (11), since $z^1_k$ must also be changed accordingly to pass the verification. Otherwise, **Adv** generates the incorrect proof $\Pi'_E = \{E'_1, E'_2\}$, $\Pi'_W$ and $\Pi'_D$ such that $E'_1$ and $E'_2$ meet the convergence condition. This can be verified by $\Pi'_W$ and $\Pi'_V$. The only feasible solution is that **Adv** trains the local model by using wrong input data, which contradicts our assumption in Section 2.3. Therefore, we can conclude that VeriDL enables the server to catch any incorrect local model update. □

## 5 EXPERIMENT

### 5.1 Setup

**Hardware & Platform.** We simulate the server on a computer of 2.10GHz CPU, 48 cores and 128GB RAM, and the participant on a computer of 2.7GHz Intel CPU and 8GB RAM. We implement both certify and verify protocols in C++. We use the implementation of bilinear mapping from PBC library[1]. The DNN algorithm is implemented in Python on TensorFlow.

**Datasets** We use two benchmark datasets for the DL community. The first is the MNIST dataset, which consists of handwritten digits images. Each image includes $28 \times 28$ pixels. The second is the TIMIT dataset that contains broadband recordings of 630 speakers. We pre-process the dataset to

[1]https://crypto.stanford.edu/pbc/.

| Dataset | # of training samples | # of validation samples | # of features | # of classes |
|---|---|---|---|---|
| MNIST | 60,000 | 6,000 | 784 | 10 |
| TIMIT | 4,620 | 462 | 100 | 61 |

**Table 3: Description of datasets**

produce 100 features for each sample. The description of these two datasets is shown in Table 3.

**Neural network architecture.** We train a DNN with four fully connected hidden layers for the MNIST dataset. We vary the number of neurons on each hidden layer from 10 to 50. We apply sigmoid function on each layer, except for the hidden layer, where we apply softmax function instead. We use stochastic gradient descent with the learning rate $\eta = 0.1$. By default, the minibatch size, i.e., the number of samples used for each round of model update, is 100. We use the same DNN structure for the TIMIT dataset with ReLU as the activation function (as suggested by the literature [17]).

**Compared approaches.** We compare the performance of VeriDL with two alternative approaches:

- $C_1$**: Homomorphic encryption vs. bilinear mapping**. When generating the proof, homomorphic encryption (HE) is used to encrypt the plaintext values in the proof instead of bilinear mapping.
- $C_2$**: Verification vs. privacy-preserving DL**. The participant encrypts the private input samples with homomorphic encryption. The server executes the learning process on the encrypted local data, and compares the computed results with the participant's uploaded updates.

For both comparisons, we use the implementation of homomorphic encryption from Helib library[2].

**Basic and optimized versions of** VeriDL. We implement two versions of VeriDL: (1) Basic approach (**B-VERIDL**): the proof of local model updates is generated for every single input example $(\vec{x}, y)$; (2) Optimized approach (**O-VERIDL**): the proof is generated for every unique value in the input $\{(\vec{x}, y)\}$. In the MNIST dataset, every image sample consists of 784 pixels, where each pixel holds a grayscale value from 0 to 255. Thus there are 256 unique feature values in total. In the TIMIT dataset, there are around $2 \times 10^8$ unique values in the input features.

## 5.2 Effectiveness of Authentication

We simulated three types of incorrect local model updates: (1) the participant sends the wrong error $E_1$ with the proof constructed from correct $E_1$; (2) the participant sends wrong $E_1$ with the proof constructed from wrong $E_1$; (3) the participant sends correct $E_1$ and wrong $E_2$. VeriDL caught all the ~~three types~~ of wrong updates with 100% guarantee.

[2]https://github.com/shaih/HElib.

## 5.3 Efficiency of Verification

We measure the performance of our approach in terms of the proof construction time, proof size, and verification time. We must note that VeriDL does not impact the accuracy of the classification result.

**Proof construction time.** In Table 4 (a), we show the proof construction time on MNIST dataset. First, we observe that O-VERIDL significantly reduces the proof construction time, compared with B-VERIDL. For instance, to generate proof for a minibatch of 500 samples, O-VERIDL only takes 5.03 seconds, while B-VERIDL demands more than 570 seconds. This is because the grayscale values (only 256 ones) repeated significantly in the MNIST dataset. Second, the time performance of B-VERIDL and O-VERIDL increases with the growth of minibatch size. Indeed, the proof construction time of both B-VERIDL and O-VERIDL increases linearly to the minibatch size. Table 4 (b) shows the result on TIMIT dataset. Due to the fact that the values in TIMIT dataset do not repeat much, the proof construction time is very similar for both B-VERIDL and O-VERIDL methods.

**Proof size.** We show the the proof size on the MNIST dataset in Figure 3 (a) and (b). Figure 3 (a) shows the proof size with regard to various minibatch size. In general, the proof size is small. Even for a minibatch with 500 samples, the total proof size of B-VERIDL is within 30 MB. Recall that for CryptoDL [10], the communication cost for a single input sample is 336 MB, while for CryptoNets [8], it is close to 600 MB. Compared with these two works, our basic approach (B-VERIDL) saves the communication overhead by 99.98%. Furthermore, the proof size of O-VERIDL is significantly smaller than B-VERIDL; it is 10% - 29% of the size by B-VERIDL. This demonstrates the advantage of O-VERIDL. Figure 3 (b) presents the proof size with regard to various number of neurons. In general, the proof size of both O-VERIDL and B-VERIDL gradually rises when the number of neurons increases. However, the growth is moderate. Therefore, VeriDL works well with large DNNs. Figure 3 (c) and (d) show the proof size for TIMIT dataset. Now the proof size for both O-VERIDL and B-VERIDL is similar, due to the fact that the values in the input features of the TIMIT dataset do not repeat much.

**Verification time.** We evaluate the verification time on both datasets, and report the results in Figure 4. We also perform the comparison $C_1$ (defined in Section 5.1). We observe that for both datasets, VeriDL (using bilinear mapping) is more efficient than using HE in the proof. Thus bilinear mapping is a good choice as it enables the same function over ciphertext with cheaper cost. Besides, the time performance of both VeriDL and HE increases when we allocate more neurons in the network. This is natural as it takes more time to verify a more complex neural network. We also notice that both

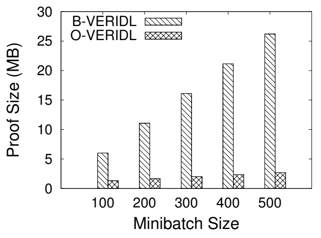
| Minibatch Size | 200 | 300 | 400 | 500 |
|---|---|---|---|---|
| B-VERIDL | 233.28 | 348.15 | 462.36 | 574.28 |
| O-VERIDL | 2.24 | 3.18 | 4.12 | 5.03 |

(a) MNIST dataset

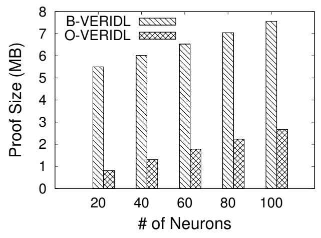
| Minibatch Size | 200 | 300 | 400 | 500 |
|---|---|---|---|---|
| B-VERIDL | 26.28 | 39.61 | 52.64 | 67.44 |
| O-VERIDL | 26.26 | 39.58 | 52.60 | 67.40 |

(b) TIMIT dataset

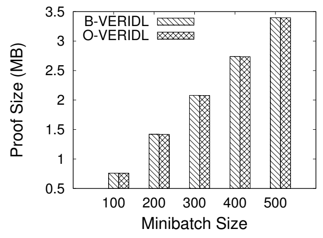
**Table 4: Proof Construction Time (seconds)**



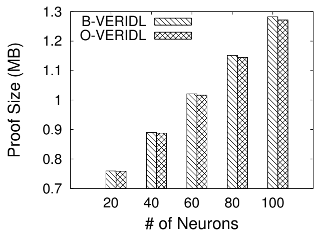
(a) Minibatch size (MNIST dataset)



(b) # of neurons (MNIST dataset)

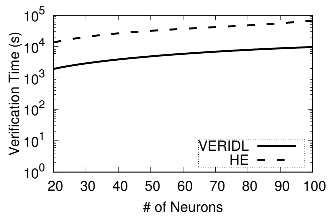


(c) Minibatch size (TIMIT dataset)

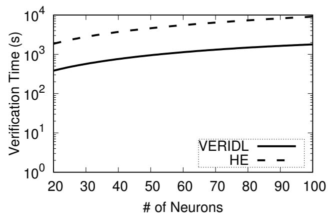


(d) # of neurons (TIMIT dataset)

**Figure 3: Proof size**

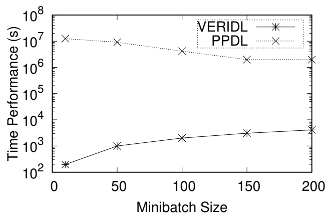


(a) MNIST dataset

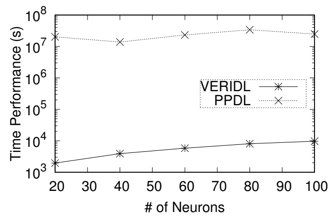


(b) TIMIT dataset

**Figure 4: Verification Time (minibatch size 100)**



(a) Minibatch size



(b) # of neurons

**Figure 5: Comparison with privacy-preserving deep learning (PPDL)**

approaches take longer time on the MNIST dataset than the TIMIT dataset. This is because the MNIST dataset includes more features; it takes the server more time to complete the verification in Formula (11) and (15).

## 5.4 Verification vs. Privacy-preserving DL

We perform the comparison $C_2$ (defined in Section 5.1) by implementing the HE-based privacy-preserving deep learning (PPDL) approaches [8, 10] and comparing the performance of VeriDL with them. To be consistent with [8, 10], we use the approximated ReLU as the activation function due to the fact that HE only supports low degree polynomials. Figure 5 shows the comparison results. In Figure 5 (a), we observe that VeriDL is faster than PPDL by more than three orders of magnitude. An interesting observation is that VeriDL and PPDL take opposite pattern of time performance when the minibatch size grows. The main reason behind the opposite pattern is that when the minibatch size grows, VeriDL has to verify $E_1$ and $E_2$ from more input samples (thus takes longer time), while PPDL needs fewer epochs to reach convergence (thus takes less time). Figure 5 (b) shows the impact of the number of neurons on the time performance of both VeriDL and PPDL. Again, VeriDL wins PPDL by at least three orders of magnitude. This demonstrates that VeriDL is more efficient than verification by PPDL.

## 6 RELATED WORK

**Verified deep learning.** The goal of *verified artificial intelligence (AI)* [21] is to design AI-based systems that are provably correct with respect to mathematically-specified requirements. Kexin Pei et al [20] designed *DeepXplore*, an automated white-box testing system for DL systems. DeepXplore generates the corner cases where DL systems may generate unexpected or incorrect behaviors. SafetyNet [7] provides a protocol that verifies the execution of DL on an untrusted cloud in the Machine-Learning-as-a-Service (MLaaS) model. The verification protocol is build on top of the *interactive proof* (IP) protocol and arithmetic circuits. The protocol only can be applied on the DNNs that can be expressed as arithmetic circuits. This places a few restrictions on DNNs, e.g., the activation functions must be polynomials with integer coefficients, which disables the activation functions that are commonly used in DNNs such as ReLU, sigmoid or softmax activations. It also requires that all the parameters and intermediate values in the network must lie in a specific integer range. Our work removes such strict assumption and can be applied to any DL system in general. He et al. also considered the integrity verification problem in the MLaaS model, and designed *VerIDeep* [9], whose key idea is to generate a few *sensitive samples* as fingerprints of DNN models.

These sensitive samples are minimally transformed inputs. They are sensitive to the change of the model in the sense that if the adversary makes only small changes to a small portion of the model parameters, the outputs of the sensitive samples from the model also change. However, VerIDeep only can provides a probabilistic correctness guarantee.

**Privacy-preserving deep learning.** The basic Federated learning framework cannot protect the privacy of the training data, even the training data is divided and stored separately [16, 23]. Shokri et. al [22] applied differential privacy (DP) [6] technique to add noise on the gradients before uploading these gradients to the server. Hitaj et. al [11] pointed out that the DP-based method in [22] failed to protect data privacy, as the adversary can learn private data through GAN (Generative Adversarial Network). Orekondy et. al [18] demonstrated the linkability attack that can link the local model updates with individual participants through the intermediate gradients. Weng et al. [24] designed a decentralized framework named *DeepChain*, which relies on blockchain-based incentive mechanism and cryptographic primitives for privacy-preserving distributed DL. Homomorphic encryption is also used to enable DL on encrypted data [8, 10].

## 7 CONCLUSION

In this paper, we design VeriDL, an integrity authentication framework that supports efficient verification of local model updates by adversarial participants in Federated learning. VeriDL extends the existing bilinear grouping technique significantly to handle the DNN model in Federated learning. The experiments demonstrate that VeriDL can verify the correctness of local model updates with cheap overhead. Several research questions remain open. First, VeriDL provides a deterministic integrity guarantee by requiring to verify the output of all neurons in DNN. An interesting direction to explore is to design an alternative *probabilistic* verification method that provides high integrity guarantee (e.g., with 95% certainty) but with much cheaper verification overhead compared with the deterministic approach. Another possible research direction is to integrate interactive proofs with VeriDL to further reduce the verification overhead.

## REFERENCES

[1] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov. How to backdoor federated learning. *arXiv preprint arXiv:1807.00459*, 2018.

[2] Y. Bengio. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[3] P. Blanchard, R. Guerraoui, J. Stainer, et al. Machine learning with adversaries: Byzantine tolerant gradient descent. In *Advances in Neural Information Processing Systems*, pages 119–129, 2017.

[4] Y. Chen, L. Su, and J. Xu. Distributed statistical machine learning in adversarial settings: Byzantine gradient descent. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(2):44, 2017.

[5] I. B. Damgård. Collision free hash functions and public key signature schemes. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 203–216, 1987.

[6] C. Dwork. Differential privacy: A survey of results. In *International Conference on Theory and Applications of Models of Computation*, pages 1–19. Springer, 2008.

[7] Z. Ghodsi, T. Gu, and S. Garg. Safetynets: Verifiable execution of deep neural networks on an untrusted cloud. In *Advances in Neural Information Processing Systems*, pages 4675–4684, 2017.

[8] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016.

[9] Z. He, T. Zhang, and R. B. Lee. Verideep: Verifying integrity of deep neural networks through sensitive-sample fingerprinting. *arXiv preprint arXiv:1808.03277*, 2018.

[10] E. Hesamifard, H. Takabi, and M. Ghasemi. Cryptodl: Deep neural networks over encrypted data. *arXiv preprint arXiv:1711.05189*, 2017.

[11] B. Hitaj, G. Ateniese, and F. Perez-Cruz. Deep models under the gan: information leakage from collaborative deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 603–618. ACM, 2017.

[12] J. Konečnỳ, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

[13] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[14] B. McMahan and D. Ramage. Federated learning: Collaborative machine learning without centralized training data. https://research.googleblog.com/2017/04/federated-learning-collaborative.html.

[15] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, et al. Communication-efficient learning of deep networks from decentralized data. *arXiv preprint arXiv:1602.05629*, 2016.

[16] L. Melis, C. Song, E. De Cristofaro, and V. Shmatikov. Inference attacks against collaborative learning. *arXiv preprint arXiv:1805.04049*, 2018.

[17] J. Michalek and J. Vanek. A survey of recent dnn architectures on the timit phone recognition task. *arXiv preprint arXiv:1806.07974*, 2018.

[18] T. Orekondy, S. J. Oh, B. Schiele, and M. Fritz. Understanding and controlling user linkability in decentralized learning. *arXiv preprint arXiv:1805.05838*, 2018.

[19] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal verification of operations on dynamic sets. In *Annual Cryptology Conference*, pages 91–110, 2011.

[20] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18. ACM, 2017.

[21] S. A. Seshia, D. Sadigh, and S. S. Sastry. Towards verified artificial intelligence. *arXiv preprint arXiv:1606.08514*, 2016.

[22] R. Shokri and V. Shmatikov. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1310–1321, 2015.

[23] C. Song, T. Ristenpart, and V. Shmatikov. Machine learning models that remember too much. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*, pages 587–601, 2017.

[24] J. Weng, J. Weng, J. Zhang, M. Li, Y. Zhang, and W. Luo. Deepchain: Auditable and privacy-preserving deep learning with blockchain-based incentive. Technical report, Cryptology ePrint Archive, Report 2018/679. 2018. Available online: https âĂę, 2018.

[25] D. Yin, Y. Chen, K. Ramchandran, and P. Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates. *arXiv preprint arXiv:1803.01498*, 2018.

## 8 APPENDIX

### 8.1 Proof of Lemma 4.1

Proof. For the sake of simplicity, we assume that $u_1$ is the only negative value in $\vec{u}$ and $\vec{v}$. It is easy to see that

$$\begin{aligned}\sum[u_i][v_i] &= [u_1][v_1] + [u_2][v_2] + \cdots + [u_m][v_m] \\ &= (u_1 + p)v_1 + u_2v_2 + \cdots + u_mv_m \\ &= z + pv_1.\end{aligned}$$

If $z \geq 0$, $[z+pv_1] = z$; otherwise, since $-p < z < p$, $[z+pv_1] = z + p$. □

### 8.2 Proof of Theorem 4.1

Proof. Let $\Delta L$ be the correct local model update, and $\Pi = \{\Pi_E, \Pi_D, \Pi_W\}$ the correct proof of $\Delta L$. Suppose an adversary **Adv** returns an incorrect local update $\Delta L' \neq \Delta L$ and the proof $\Pi' = \{\Pi'_E, \Pi'_D, \Pi'_W\}$. We assume that **Adv** is fully aware of the details of our authentication method. Thus it will try to generate $\Pi'$ intelligently, trying to escape from the authentication. According to Formula (15), $\Pi_W$ and $\Pi_D$ must be consistent in order to pass the verification. There are three types of possible adversarial behaviors by **Adv**: (1) $\Pi'_E = \Pi_E$, $\Pi'_D = \Pi_D$, and $\Pi'_W = \Pi_W$ (i.e., correct errors and correct proof); (2) $\Pi'_E \neq \Pi_E, \Pi'_D = \Pi_D, \Pi'_W = \Pi_W$ (i.e., incorrect errors and correct proof); and (3) $\Pi'_E \neq \Pi_E, \Pi'_D \neq \Pi_D, \Pi'_W \neq \Pi_W$ (i.e., incorrect errors and incorrect proofs). Next, we discuss these behaviors case by case, and show that for each case, the incorrect local update can escape from the authentication with zero probability (Def. 2.2).

**Case 1: $\Pi'_E = \Pi_E$, $\Pi'_D = \Pi_D$, and $\Pi'_W = \Pi_W$.** For this case, the verification by Formula (11) must fail, since $z^1_k$ is generated by using the weights with correct update. It cannot match with $z^1_k$ by using incorrect weights.

**Case 2: $\Pi'_E \neq \Pi_E, \Pi'_D = \Pi_D$, and $\Pi'_W = \Pi_W$.** The adversary **Adv** returns incorrect $\Pi'_E = \{E'_1, E'_2\}$, where either $E'_1 \neq E_1$ or $E'_2 \neq E_2$, and the correct proof $\Pi_D$ constructed from the right $E_1$ and $E_2$. Then **Adv** can pass the verification in Formula (11), since in $\Pi_D$, $z^1_k = \sum_{x_i w^1_{ik}}$. However, if $E'_1 \neq E_1$, it can be detected with 100% probability due to the fact that $E'_1$ must fail the verification in Formula (12). If $E'_1 = E_1$ and $E'_2 \neq E_2$, After the server updates the weight parameters and calculates the output, he is able to spot the erroneous $E'_2$ because $E'_2$ cannot pass the verification in Formula (12).

**Case 3: $\Pi'_E \neq \Pi_E, \Pi'_D \neq \Pi_D, \Pi'_W \neq \Pi_W$.** The adversary **Adv** forges the proof $\Pi'_D$, aiming to let $\Pi'_E$ pass the verification. The proof $\Pi'_D$ and $\Pi'_W$ must satisfy two conditions: (1) $E'_1$ and $E'_2$ meet the convergence condition; and (2) $E'_1$, $E'_2$ and $\Pi'_D$ and $\Pi'_W$ pass the verification in Formula (11 - 15). Recall that $\Pi'_D = \{\{g^{x'_i}\}, g^{y'}, \{z^{1'}_k\}, g^{\delta^{o'}}, \{\delta^{L'}_k\} | \forall(\vec{x}, y) \in D\}$, and $\Pi'_W = \{\Delta w^{1'}_{jk}\}$. There are four possible approaches that **Adv** can launch to forge $\Pi'_V$.

**Case 3.1**: $z^{1'}_k \neq z^1_k$ or $\hat{z}^{1'}_k \neq \hat{z}^1_k$, for some $1 \leq k \leq d_1$. In this case, **Adv** provides incorrect linear activation on the first hidden layer of the DNN. With access to $\{g^{x_i}\}$, the server is capable of identifying this cheating behavior because $z^{1'}_k$ and $\hat{z}^{1'}_k$ cannot pass the verification by Formula (11).

**Case 3.2**: $g^{\delta^{o'}} \neq g^{\delta^o}$. Any incorrect error signal on the output layer must be detected. This is because the server learns $o$ and $z^o$ in the first feedforward procedure. As a consequence, according to the biliearity property of bilinear pairing (Section 2.1), the incorrect error signal must fail the verification by Formula (13).

**Case 3.3**: $\delta^{L'}_k \neq \delta^L_k$ for some $1 \leq k \leq d_L$. Similar to the analysis in Case 3.3, during the first feedforward computation, the server computes $(z^{L-1}_k$. Therefore, if $\delta^{L'}_k \neq \delta^L_k$, it must fail the verification by Formula (14).

**Case 3.4**: $\Delta w^{1'}_{jk} \neq \Delta w^1_{jk}$ for some $1 \leq j \leq m$ and $1 \leq k \leq d_1$. Since the server computes the correct error signal $\{\delta^1_k\}$ and obtains $\{g^{x_j}\}$ from $\Pi'_D$, the incorrect weight increment will fail the verification by Formula (15).

**Case 3.5**: $g^{x'_i} \neq g^{x_i}$ for some $i$ or $g^{y'} \neq g^y$. Because **Adv** is not allowed to manipulate $x_i$ or $y$, he can only cheat on the generator $g$. In particular, **Adv** uses a different generator $h$ to produce $h^{x_i}$ and $h^y$. Obviously, $h = g^q$, where $gcd(p, q) = 1$, and $p$ is the order of the cyclic group $G$. Given $\Pi e(h^{x_i}, g^{w^1_{ik}}) = e(g^{qx_i}, g^{w^1_{ik}}) = e(g, g)^{z^1_k q}$, in order to pass the verification in Formula (11), **Adv** must set $z^{1'}_k = z^1_k q$. However, manipulating the linear activation of the first hidden layer completely changes the behavior of the neural network, including the number of epochs to reach convergence, the output and error at convergence. Therefore **Adv** cannot come up with $E'_1$, $E'_2$ and $\{\frac{\partial C}{\partial w^1_{jk}}'\}$ to pass the verification in Formula (12) and (15) without executing the training process. For example, **Adv** can do following. First, it generates $r, s \in Z_p$ s.t. $gcd(r, p) = 1$. Second, it creates the fake validation data $\vec{x}' = r\vec{x}$ and $y' = sy$. Third, it trains the DNN with the initial weight parameters $W$ and fake validation data $(\vec{x}', y')$. It obtains two consecutive validation error $E'_1$ and $E'_2$ at convergence, as well as the weight update $\Delta L'$. Finally, it constructs the proof $\Pi'_E = \{g^{E'_1}, g^{E'_2}\}$, and $\Pi'_D = \{\{h^{x_i}_1\}, h^y_2, \{z^{1'}_k\}, \{\hat{z}^{1'}_k\}, g^{\delta^{o'}}, \{\delta^{L'}_k\} | \forall(\vec{x}, y) \in V\}$ and $\Pi_W = \{\Delta w^{1'}_{jk}\}$, where $h_1 = g^r$, $h_2 = g^s$, $z^{1'}_k = \sum x'_i w^1_{ik}$, $\{\Delta w^{1'}_{jk}\}$, $\{\delta^{o'}_k\}$ and $\{\delta^{L'}_k\}$ are weight increment and the error signals at convergence with $\{\vec{x}', \vec{y}'\}$. Obviously, $\Delta L'$ and $\Pi'$

can pass the verification process. However, we argue that this attack is equivalent to the process that **Adv** uses incorrect input data $\{\vec{x}', y'\}$. This violates our assumption that the participant cannot cheat on data integrity. Thus, such attack can be caught by data integrity verification.

With all the cases discussed, we conclude that VeriDL is secure. $\square$