TrustDDL: A Privacy-Preserving Byzantine-Robust Distributed Deep Learning Framework

René Klaus Nikiel
TU Darmstadt
Darmstadt, Germany
rene.nikiel@gmail.com

Meghdad Mirabi
DFKI & TU Darmstadt
Darmstadt, Germany
meghdad.mirabi@dfki.de

Carsten Binnig
TU Darmstadt & DFKI
Darmstadt, Germany
carsten.binnig@cs.tu-darmstadt.de

***Abstract*—This paper introduces a distributed deep learning framework called TrustDDL crafted to address privacy and Byzantine robustness concerns across the training and inference phases of deep learning models. The framework incorporates additive secret-sharing-based protocols, a *commitment* phase, and redundant computation to identify *Byzantine* parties and shield the system from their detrimental effects during both deep learning model training and inference. It ensures uninterrupted protocol execution, guaranteeing reliable output delivery in both phases. Our security analysis affirms the efficacy of the proposed framework against both *honest-but-curious* and *malicious* adversaries for learning and inference tasks. Furthermore, we evaluate the proposed framework against existing open-source distributed machine learning frameworks, underscoring its practicality for developing and deploying distributed deep learning systems.**

***Index Terms*—Byzantine Robustness, Computational Redundancy, Deep Learning, Privacy Preserving, Secret Sharing**

I. Introduction

Deep learning, a leading methodology in machine learning, excels in modeling and recognizing complex data types such as images, speech, and text [1], [2]. Despite hardware and network advancements, large-scale deep learning tasks remain computationally intensive, posing challenges for resource-constrained systems [3], [4]. Distributed deep learning frameworks have emerged to address these challenges, alongside commercial "Deep Learning as a Service" solutions offered by cloud providers [5]–[7]. However, this approach brings its own set of challenges that warrant careful consideration.

One major concern is the high risk of privacy breaches, mainly arising from both the input data and the deep learning model itself [8], [9]. Input data frequently contains sensitive information, requiring protection during transit and processing. Likewise, deep learning models, as valuable assets, must be shielded from theft, tampering, or unauthorized access.

In addition, distributed deep learning is particularly vulnerable to the presence of "misbehaving" parties within the system [10], [11], known as *Byzantine* parties. Such deviations can result from software and hardware bugs, the inclusion of poisoned or compromised data [12], [13], or the actions of malicious parties attempting to manipulate the system [14].

Several cryptographic-based approaches, including Interactive Proof Systems [15], [16], Homomorphic Encryption [17], [18], and Zero-Knowledge Proofs [19], [20], have been proposed for preserving the privacy of sensitive data and verifying computation results during both model training and inference phases. However, these approaches often incur significant computational and communication overhead. Alternatively, leveraging Trusted Execution Environments (TEEs) offers another solution to protect data and model privacy and integrity. Yet, TEE-based solutions may face challenges such as malware attacks [21] and side-channel attacks [22].

Recently, secret sharing schemes have been explored for designing efficient privacy-preserving machine learning frameworks [1], [2], [23], [24]. However, these schemes, which assume an *honest-but-curious* adversary model, lack resistance against a *malicious* adversary, representing a form of *Byzantine* party.

Several *Byzantine* machine learning frameworks have been proposed, primarily focusing on server-based coordination mechanisms, coordinating parallel model training tasks through parameter servers [25], [26]. This includes federated learning, where models are trained across different devices without exchanging data samples. In contrast, peer-to-peer coordination mechanisms [27], [28] involve coordinating deep learning tasks through direct interaction between computing parties. However, existing *Byzantine* machine learning approaches differ from the assumption in this paper, where a client lacks the capacity for performing deep learning tasks and outsources storage and computational requirements to untrusted cloud service providers. The client must ensure data privacy and system robustness against *Byzantine* failures.

To tackle the limitations of existing approaches, this paper proposes TrustDDL, a distributed deep learning framework that addresses privacy and Byzantine robustness concerns. TrustDDL secures all computations during model training and inference phases using additive secret sharing and data masking techniques. Additionally, it employs computational redundancy and robust confirmation methods to filter out incorrect intermediate results returned by *Byzantine* parties and to impart robustness against arbitrary actions of *Byzantine* parties. To achieve this, TrustDDL decomposes operations involved in deep learning into simple arithmetic operations such as addition and multiplication, which can be computed efficiently over additive secret-shared data in a privacy-preserving manner. TrustDDL includes a *commitment* phase coupled with additive secret-sharing protocols to detect *Byzantine* parties, allowing it to discard incorrect intermediate results during

the training and inference of a deep learning model. The redundant computation employed in TrustDDL also provides robustness against arbitrary actions of *Byzantine* parties. We define *Byzantine* robustness as the ability to recover from *Byzantine* failures and continue learning or inference tasks without having to abort within the proposed protocols — a property also known as *guaranteed output delivery*. The main contributions of this paper are as follows:

- We propose TrustDDL to address privacy and Byzantine robustness in deep learning training and inference. Through redundant computations and a *commitment* phase, TrustDDL effectively detects and recover from *Byzantine* failures.
- We demonstrate, through our security analysis detailed in the appendix, that TrustDDL effectively guards against both *honest-but-curious* and *malicious* adversary models. TrustDDL stands out as a secret-sharing-based solution capable of sustaining continuous learning and inference tasks without protocol interruptions.
- We benchmark TrustDDL against existing open-source distributed secret-sharing-based machine learning frameworks for deep learning model training and inference.

II. Preliminaries

The Additive Secret Sharing scheme (ASS) is the fundamental building block used in TrustDDL. In ASS, the *secret*, e.g., s, is split into *N* different *additive secret shares* $[s]_i$ for $\forall i = 1, \ldots, N$, satisfying $\sum_{i=1}^{N}[s]_i = s$, where $s \in \mathbb{R}$. It is an (N, N)-threshold secret sharing scheme as the absence of any additive secret share renders the information independent from the secret. Additive secret shares can be created by choosing $N-1$ random shares $[s]_1, \ldots, [s]_{N-1}$ and setting $[s]_N = s - \sum_{i=1}^{N-1}[s]_i$. Algorithm 1 shows this process in pseudo-code form for a secret $s \in \mathbb{R}^{m\times n}$. For simplicity, we define all protocols presented in this paper over the Ring of real matrices $\mathbb{R}^{m\times n}$, with the case $n = m = 1$ representing the set of real numbers.

Algorithm 1: *CreateShares*(s, N)

Input: A secret value $s \in \mathbb{R}^{m\times n}$;
Output: Additive secret shares $[s]_1, ..., [s]_N$ of s;
for $i = 1, \ldots, (N-1)$ **do**
  Generate a random number $r_i \in \mathbb{R}^{m\times n}$;
  $[s]_i \leftarrow r_i$;
$[s]_N \leftarrow s - \sum_{i=1}^{N-1}[s]_i$;
return $[s]_1, ..., [s]_N$;

ASS supports homomorphic addition and subtraction by definition. Consider the case of N parties, where each party i holds additive secret shares $[x]_i, [y]_i \in \mathbb{R}^{m\times n}$ for two secrets $x, y \in \mathbb{R}^{m\times n}$. From now on, the term *share* will be used interchangeably with *additive secret share*, as this is the technique of secret sharing used in TrustDDL. To obtain a share $[z]_i \in \mathbb{R}^{m\times n}$ representing the result of the addition $z = x \pm y$, each party i can locally compute $[z]_i = [x]_i \pm [y]_i$.

Furthermore, multiplication and division by constants are supported by ASS. From this point on, we denote a regular multiplication by the symbol "$\cdot$", matrix multiplication by "$\times$", and a division by "$\div$". For the operators $* \in \{\cdot, \div\}$, each party i can locally compute $[z]_i = [x]_i * k$, where $[x]_i$ is a share of secret x and k is a constant, to obtain a share of $z = x * k$.

To support regular multiplication and matrix multiplication between two secrets, ASS generates the *Beaver triple* [29] $\{a, b, c | c = ab\}$, where a and b are randomly chosen from $\mathbb{R}$ such that $c = ab$. This triple must be prepared for each multiplication. The triple is then split into triples of shares $([a]_i, [b]_i, [c]_i)$ by a trusted party using Algorithm 1 and distributed among the N untrusted parties. Using this triple, party i can mask the shares $[x]_i$ and $[y]_i$ by setting $[e]_i = [x]_i - [a]_i$ and $[f]_i = [y]_i - [b]_i$. Masking the original shares by adding or subtracting random values preserves the privacy of the secrets x and y; even if all shares $[e]_i$ and $[f]_i$ for $\forall i = 1, \ldots, N$ are known, the secrets cannot be reconstructed [30]. Thus, all parties can exchange their values $[e]_i$ and $[f]_i$ so that each party obtains all shares $[e]_i$ and $[f]_i$ for $\forall i = 1, \ldots, N$, without leaking any information about x and y. The parties can then reconstruct $e = \sum_{i=1}^{N}[e]_i$ and $f = \sum_{i=1}^{N}[f]_i$. After reconstructing e and f, each party i sets $[z]_i = [c]_i + e * [b]_i + [a]_i * f$, where $* \in \{\cdot, \times\}$. Finally, one random party r, $1 \le r \le N$, sets $[z]_r = [z]_r + e * f$ [31].

This approach supports various types of privacy-preserving multiplication: regular multiplication with $a, b \in \mathbb{R}$, element-wise multiplication with $a, b \in \mathbb{R}^{m\times n}$, and matrix multiplication with $a \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^{n\times p}$. Algorithm 2 (*SecMul*) demonstrates element-wise multiplication, while *SecMatMul* can be obtained by substituting element-wise with matrix multiplications. For brevity, we omit the explicit presentation of this protocol. Instead of collecting all shares $[e]_i$ and $[f]_i$ for $\forall i = 1, \ldots, N$ at each party, it is possible to collect these shares at one random designated party r, reconstruct e and f there, and distribute the results to the rest of the parties. This optimization, utilized in Algorithm 2, significantly reduces communication costs.

Algorithm 2: *SecMul*$([x]_i, [y]_i, B_i, i, r)$

Input: Shares $[x]_i, [y]_i \in \mathbb{R}^{m\times n}$ of the matrices to multiply element-wise, a Beaver triple $B_i = ([a]_i, [b]_i, [c]_i) \in \mathbb{R}^{m\times n} \times \mathbb{R}^{m\times n} \times \mathbb{R}^{m\times n}$, the number of the party calling this protocol $i \in \{1, ..., N\}$, and the randomly selected party r;
Output: A share $[z]_i \in \mathbb{R}^{m\times n}$ of the result of the multiplication $z = x \cdot y$;
$[e]_i \leftarrow [x]_i - [a]_i$;
$[f]_i \leftarrow [y]_i - [b]_i$;
if $i == r$ **then**
  Receive $[e]_j$ and $[f]_j$ from all parties $j \in \{1, ..., N\}\backslash\{i\}$;
  Recover $e = \sum_{j=1}^{N}[e]_j$ and $f = \sum_{j=1}^{N}[f]_j$;
  Send e and f to all parties $j \in \{1, ..., N\}\backslash\{i\}$;
  $[z]_i \leftarrow [c]_i + e \cdot [b]_i + [a]_i \cdot f + e \cdot f$;
else
  Send $[e]_i$ and $[f]_i$ to party r;
  Receive e and f from party r;
  $[z]_i \leftarrow [c]_i + e \cdot [b]_i + [a]_i \cdot f$;
return $[z]_i$;

To compare two secrets $x, y \in \mathbb{R}^{m\times n}$ element-wise, it suffices to determine the sign of $x-y$, denoted as $sign(x-y)$ [30]. Since the reconstruction of $x-y$ leaks information about x and y, we need to reconstruct $t \cdot (x-y)$ instead, where $t \in \mathbb{R}^{m\times n}$

with random positive elements. Choosing t to be positive for all elements ensures $sign(t \cdot (x-y)) = sign(x-y)$ [31]. With shares $[x]_i, [y]_i$, and $[t]_i$ for x, y, and t, each party i calculates $[\alpha]_i = [x]_i - [y]_i$ and obtains a share of $t \cdot (x-y) = t \cdot \alpha$ as $[\beta]_i = \textit{SecMul}([t]_i, [\alpha]_i, B_i, i, r)$. Similar to the *SecMul* protocol, one selected party r collects all shares $[\beta]_i$ for $\forall i = 1, ..., N$, reconstructs β, and sends it to all other parties. Each party then obtains the result as $sign(x-y)$, which is equivalent to $sign(\beta)$. This approach is used in Algorithm 3 for privacy-preserving comparison between two secrets.

Algorithm 3: $\textit{SecComp}([x]_i, [y]_i, [t]_i, B_i, i, r)$

Input: Shares $[x]_i, [y]_i \in \mathbb{R}^{m \times n}$ of the matrices to compare element-wise, an auxiliary share $[t]_i \in \mathbb{R}^{m \times n}$, a Beaver triple $B_i = ([a]_i, [b]_i, [c]_i) \in \mathbb{R}^{m \times n} \times \mathbb{R}^{m \times n} \times \mathbb{R}^{m \times n}$, the number of the party calling this protocol $i \in \{1, ..., N\}$, and the randomly selected party r;

Output: The sign of $x - y$, $sign(x-y)$;

$[\alpha]_i \leftarrow [x]_i - [y]_i$;
$[\beta]_i \leftarrow \textit{SecMul}([t]_i, [\alpha]_i, B_i, i, r)$;
if $i == r$ **then**
  Receive $[\beta]_j$ from all parties $j \in \{1, ..., N\} \backslash \{i\}$;
  Recover $\beta = \sum_{j=1}^{N} [\beta]_j$;
  Send β to all parties $j \in \{1, ..., N\} \backslash \{i\}$;
else
  Send $[\beta]_i$ to party r;
  Receive β from party r;
$sign(x-y) \leftarrow sign(\beta)$;
return $sign(x-y)$;

III. Proposed Framework

A. System Architecture

TrustDDL's system architecture, depicted in Fig. 1, consists of a proxy layer with three computing parties collaborating on most operations involved in mode training and inference. This layer acts as an intermediary between the data owner and the model owner. In TrustDDL, the roles of both data owner and model owner are consolidated into a single trusted party that leverages TrustDDL's computational and storage capabilities for deep learning. In TrustDDL, the following actors come into play:

- *Data Owner*: A party who holds the training and testing data and creates the three sets of shares for all inputs and labels. This party also may receive the results of an inference task, i.e., the predicted label.
- *Model Owner*: A party who possesses the (deep) neural network to be trained or used for inference. This party is responsible for creating and distributing shares for neural network's parameters as well as auxiliary values (e.g., Beaver triples and auxiliary positive numbers) and their relative shares. This party may also receive the result of a training task, i.e., the updated model parameters.
- *Computing Parties*: Computing parties in the proxy layer perform most of the operations involved in model training and inference, validate computation results, and aggregate them when necessary.

This framework, tailored for $N = 2$ and three computing parties (a 3PC framework), accommodates at most one potential *Byzantine* party in the proxy layer. In TrustDDL, either

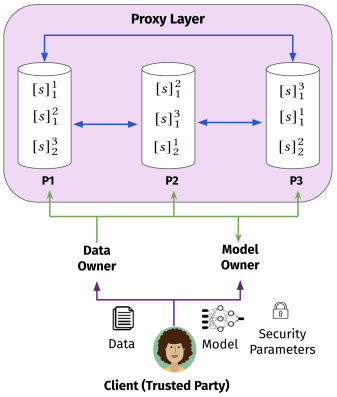


Fig. 1: System Architecture

the data owner or the model owner generates three distinct sets of shares for each secret, ensuring that no computing party in the proxy layer obtains a complete set. For any secret s, the data owner or the model owner creates the three sets of shares $s^1 = \{[s]_1^1, [s]_2^1\}, s^2 = \{[s]_1^2, [s]_2^2\}, s^3 = \{[s]_1^3, [s]_2^3\}$, where $[s]_i^j$ represents the ith share of the jth set. In TrustDDL, party P_1 receives the shares $\{[s]_1^1, [s]_1^2, [s]_2^3\}$, P_2 receives $\{[s]_1^2, [s]_1^3, [s]_2^1\}$ and P_3 receives $\{[s]_1^3, [s]_1^1, [s]_2^2\}$. The distribution scheme in TrustDDL is designed so that it meets the requirements of privacy—ensuring that no single computing party obtains a complete set of N shares—and resiliency—where 2 out of 3 computing parties are sufficient to perform computations involved in deep learning tasks, even in the presence of one *Byzantine* party in the proxy layer.

B. Resiliency against a Byzantine Party

TrustDDL ensures resilience against a *Byzantine* party in the proxy layer by redundantly executing each ASS-based protocol across all computing parties. This includes protocols like matrix multiplication and comparison, where each party collects shares from the others for secure reconstructions. This method allows for the high probability identification of a *Byzantine* party.

Here, we go through one exemplary reconstruction phase, derive a decision rule, and argue for its correctness. We focus on the reconstructions in one specific party (P_1); however, the correctness can be argued for any other computing party in the same way. Party P_1 may need to use a timer to handle potential delays or dropped shares from parties P_2 and P_3. For simplicity, we focus on detecting a *Byzantine* party in the proxy layer, presuming all shares are received by party P_1.

When party P_1 reconstructs some value s, it holds the shares $\{[s]_1^1, \widehat{[s]_1^2}, [s]_2^3\}$ and collects the shares $\{[s]_1^2, \widehat{[s]_1^3}, [s]_2^1\}$ of party P_2 and $\{[s]_1^3, \widehat{[s]_1^1}, [s]_2^2\}$ of party P_3. We introduce the notation $\widehat{\cdot}$ to avoid duplicate names, and this notation is used in all subsequent protocols. It then performs the following reconstructions:

$$s^1 = [s]_1^1 + [s]_2^1, \quad s^2 = [s]_1^2 + [s]_2^2, \quad s^3 = [s]_1^3 + [s]_2^3,$$
$$\widehat{s^1} = \widehat{[s]_1^1} + [s]_2^1, \quad \widehat{s^2} = \widehat{[s]_1^2} + [s]_2^2, \quad \widehat{s^3} = \widehat{[s]_1^3} + [s]_2^3$$

In this case, compromised shares of one computing party (P_2 or P_3) can only corrupt the set of reconstructions $\{s^2, \widehat{s^3}, s^1, \widehat{s^1}\}$ or $\{s^3, \widehat{s^1}, s^2, \widehat{s^2}\}$. Thus, if party P_2 is a *Byzantine* party, the reconstructions $\widehat{s^2}$ and s^3 are always correct. Similarly, if party P_3 is a *Byzantine* party, the reconstructions s^1 and $\widehat{s^3}$ are always correct.

In addition, to prevent a *Byzantine* party from reliably selecting incorrect shares that closely match corrupted reconstructions, TrustDDL includes a *commitment* phase in ASS-based protocols. Here, computing parties commit to their shares before distributing them to others, exchanging them only after receiving commitment values (i.e., hash values of shares) from the others. Subsequently, each computing party recalculates the hash values and verifies whether they match the values received during the *commitment* phase.

In the case where a *Byzantine* party violates the *commitment* phase by sending a hash value of one share but later exchanging it with a different share, the two honest computing parties can detect this and discard any reconstruction involving the *Byzantine* party's shares. However, if it only violates the *commitment* phase with one party, the two honest computing parties are unable to reach a consensus on which computing party has violated the *commitment* phase. Nevertheless, this does not hinder correct reconstructions, as TrustDDL's computing parties independently detect and recover from misbehavior.

There is another case where a *Byzantine* party adheres to the *commitment* phase but uses an incorrect share in both the calculation of the hash value and in the share exchange. This misbehavior cannot be detected by recalculating the hash values and verifying whether they match or not after the *commitment* phase. To delve deeply into this case, assume that there is a *Byzantine* party P_i; it sends a set of shares $\{[s]_1^{i_1}, \widehat{[s]_1^{i_2}}, [s]_2^{i_3}\}$ (where i_1, i_2, and i_3 denote the sets of which P_i holds its shares, e.g., $i_1 = 2$, $i_2 = 3$, and $i_3 = 1$ for party P_2). With these shares, it can corrupt the reconstructions s^{i_1}, $\widehat{s^{i_2}}$, s^{i_3}, and $\widehat{s^{i_3}}$. For these reconstructions, we have $s^{i_3} \approx \widehat{s^{i_3}}$ (in this context, we write '$\approx$' instead of '$=$' since the reconstruction of a secret from two different sets of shares incurs a slight inaccuracy when implemented with the finite precision of a computer). Thus, we ignore the (approximate) equality between any two reconstructions s^j and $\widehat{s^k}$ if $j = k$. However, the *Byzantine* party P_i cannot force an (approximate) equality between s^{i_1} and $\widehat{s^{i_2}}$ without the knowledge of the other shares of the respective sets ($[s]_2^{i_1}$ and $[s]_2^{i_2}$). Due to the existence of *commitment* phase in ASS-based protocols of TrustDDL, the *Byzantine* party P_i cannot simply wait to receive these shares and then send incorrect shares to force (approximate) equality since it has to commit to its shares beforehand. Thus, if the *Byzantine* party P_i sends incorrect shares, it only has a negligible probability of randomly achieving an (approximate) equality between s^{i_1} and $\widehat{s^{i_2}}$. Since the remaining two correct reconstructions are always (approximately) equal, the honest computing parties can identify two correct reconstructions by determining the minimum distance between any pair of two reconstructions $s^j, \widehat{s^k}$, where $j \neq k$:

$$\min_{s^j, \widehat{s^k}} \{dist(s^j, \widehat{s^k}) \mid j \neq k\}_{j,k \in \{1,2,3\}}$$

for the distance measure *dist*. This serves as a decision rule in TrustDDL to identify the correct computations.

TrustDDL applies such a strategy in ASS-based protocols to provide resilience against one *Byzantine* party, as illustrated in Algorithm 4 and Algorithm 5.

In Algorithm 4 (*SecMul-BT* protocol), each party i contributes all its shares of the matrices to be multiplied, represented as vectors: $[\mathbf{x}]_i = ([x]_1^{i_1}, \widehat{[x]_1^{i_2}}, [x]_2^{i_3})$, $[\mathbf{y}]_i = ([y]_1^{i_1}, \widehat{[y]_1^{i_2}}, [y]_2^{i_3})$, each consisting of three shares. Additionally, party i provides its shares of the Beaver triple as $[\mathbf{B}]_i = (\widehat{[\mathbf{a}]_i}, [\mathbf{b}]_i, [\mathbf{c}]_i)$, where $[\mathbf{a}]_i = (\widehat{[a]_1^{i_1}}, [a]_1^{i_2}, [a]_2^{i_3})$, $[\mathbf{b}]_i = ([b]_1^{i_1}, \widehat{[b]_1^{i_2}}, [b]_2^{i_3})$ and $[\mathbf{c}]_i = ([c]_1^{i_1}, \widehat{[c]_1^{i_2}}, [c]_2^{i_3})$, with each also being a vector of three shares. The indices i_1, i_2, and i_3 indicate the set from which the share originates (e.g., $i_1 = 1$, $i_2 = 2$, $i_3 = 3$ for party P_1; see Fig. 1). For simplicity, we implicitly use these values within Algorithm 4 without passing them as arguments. Furthermore, this algorithm can be applied for any share dimensions to perform an element-wise multiplication over $\mathbb{R}^{m \times n}$. We omit the dimensions of all shares here for the sake of readability. Following this, each computing party calculates vectors of the intermediate shares $[\mathbf{e}]_i$ and $[\mathbf{f}]_i$ by performing element-wise subtractions between two vectors (Lines 1 and 2). Subsequently, all computing parties execute the *commitment* phase by exchanging the hash values of intermediate shares (Lines 3-7). To address potential issues such as delayed or dropped messages from a *Byzantine* party during the *commitment* phase, or a *malicious* party falsely claiming not to have received any commitment values, it is possible to introduce timeouts. Computing parties can forward the commitment values of others in response. If a *Byzantine* party deliberately delays or drops all of its messages, the other two parties can reach a consensus on this misbehavior and exclude the offending party from further computations. In the case of a *Byzantine* party only delaying or dropping their shares to one party, the other computing party can forward the received commitment values (directly or after a timeout). For simplicity, we do not detail this mechanism in Algorithm 4. Only after confirming the receipt of all commitment values (Line 8) do the computing parties proceed to exchange their intermediate shares. During the exchange of all intermediate shares (Lines 9-14), the computing parties send their intermediate shares (Line 10) and store the intermediate shares received by the other two parties (Line 11). Furthermore, they recalculate the hash values for all received intermediate shares and check if they match the hash values received in the *commitment* phase (Line 12). For any reconstructions s^j and $\widehat{s^j}$, this algorithm uses the flags $flag_j$ and $\widehat{flag_j}$ to indicate whether all intermediate shares used in the reconstruction were validated in the *commitment* phase (value *true*) or not (value *false*). When party P_i receives some intermediate shares $[\mathbf{s}]_j =$

$([s]_1^{j_1}, \widehat{[s]_1^{j_2}}, [s]_2^{j_3})$ from party P_j (here, s would be e or f), this algorithm first initializes the flags of reconstructions affected by a violation of the *commitment* phase by party P_j: flag_{j_1}, $\widehat{\mathit{flag}}_{j_2}$, flag_{j_3}, and $\widehat{\mathit{flag}}_{j_3}$. This initialization is abstracted to *initializeFlags*, which sets each passed argument to *true* if they are undefined or *identity* otherwise (Line 13). Then, these flags are updated by performing an element-wise *AND* operation with the commit check (Line 14). Thus, each flag is only *true* if the commit check succeeded for all intermediate shares of the corresponding reconstruction set. Next, reconstructions flagged as *Byzantine* (value *false*) are ignored. Next, each computing party performs six reconstructions for e and f (Lines 15-19). The algorithm identifies two correct reconstructions from each of them by determining the minimum distance between any pair of two reconstructions (Line 20) and dismisses any reconstructions that have been flagged due to a violation of the *commitment* phase. Finally, each computing party calculates and returns its shares of $z = x \cdot y$ for the sets (and share numbers within a set) they were assigned in the system architecture (Lines 21-25). In Algorithm 4, r is set to 2, signifying the addition of the term $e \cdot f$ to the second share of each set (Line 23). Consequently, there is no designated party P_r to perform different operations, as observed in Algorithm 2 (*SecMul* protocol) in Section II. Thus, a fixed value for r can be chosen within the share calculation rules.

The adapted *SecMatMul-BT* protocol is derived from Algorithm 4 by substituting element-wise multiplications with matrix multiplications. Due to space constraints, we do not explicitly show this protocol. However, we provide the adapted *SecComp* protocol in Algorithm 5, following a similar strategy without further explanation.

Algorithm 4: *SecMul-BT* $([\mathbf{x}]_i, [\mathbf{y}]_i, [\mathbf{B}]_i, i)$

Input: Shares $[\mathbf{x}]_i = ([x]_1^{i_1}, \widehat{[x]_1^{i_2}}, [x]_2^{i_3})$ and $[\mathbf{y}]_i = ([y]_1^{i_1}, \widehat{[y]_1^{i_2}}, [y]_2^{i_3})$ of the matrices to multiply element-wise (i_1, i_2, and i_3 denote the sets of which party P_i holds shares), shares of a Beaver triple $[\mathbf{B}]_i = ([\mathbf{a}]_i, [\mathbf{b}]_i, [\mathbf{c}]_i)$ where $[\mathbf{a}]_i = ([a]_1^{i_1}, \widehat{[a]_1^{i_2}}, [a]_2^{i_3})$, $[\mathbf{b}]_i = ([b]_1^{i_1}, \widehat{[b]_1^{i_2}}, [b]_2^{i_3})$ and $[\mathbf{c}]_i = ([c]_1^{i_1}, \widehat{[c]_1^{i_2}}, [c]_2^{i_3})$, and the number i of the party calling this protocol;

Output: Shares $[\mathbf{z}]_i = ([z]_1^{i_1}, \widehat{[z]_1^{i_2}}, [z]_2^{i_3})$ of the result of the multiplication $z = x \cdot y$;

1 $[\mathbf{e}]_i \leftarrow [\mathbf{x}]_i - [\mathbf{a}]_i$;
2 $[\mathbf{f}]_i \leftarrow [\mathbf{y}]_i - [\mathbf{b}]_i$;
3 **for** $i, j \in \{1, 2, 3\}$, $i \neq j$ **do**
4 Send $\mathit{hash}([\mathbf{e}]_i), \mathit{hash}([\mathbf{f}]_i)$ to P_j;
5 Receive $\mathit{hash}([\mathbf{e}]_j), \mathit{hash}([\mathbf{f}]_j)$ from P_j;
6 $\mathit{hash}_{e,j} \leftarrow \mathit{hash}([\mathbf{e}]_j)$;
7 $\mathit{hash}_{f,j} \leftarrow \mathit{hash}([\mathbf{f}]_j)$;
8 Confirm receipt of all the commitment values from all other parties;
9 **for** $i, j \in \{1, 2, 3\}$, $i \neq j$ **do**
10 Send $[\mathbf{e}]_i, [\mathbf{f}]_i$ to P_j;
11 Receive and store $[\mathbf{e}]_j = ([e]_1^{j_1}, \widehat{[e]_1^{j_2}}, [e]_2^{j_3})$ and $[\mathbf{f}]_j = ([f]_1^{j_1}, \widehat{[f]_1^{j_2}}, [f]_2^{j_3})$ from P_j;
12 $\mathit{commit\_check}_j \leftarrow \mathit{hash}_{e,j} == \mathit{hash}([\mathbf{e}]_j) \wedge \mathit{hash}_{f,j} == \mathit{hash}([\mathbf{f}]_j)$;
13 *initializeFlags*($\mathit{flag}_{j_1}, \widehat{\mathit{flag}}_{j_2}, \mathit{flag}_{j_3}, \widehat{\mathit{flag}}_{j_3}$);
14 $\mathit{flag}_{j_1}, \widehat{\mathit{flag}}_{j_2}, \mathit{flag}_{j_3}, \widehat{\mathit{flag}}_{j_3} \overset{\wedge}{\leftarrow} \mathit{commit\_check}_j$;
15 **for** $j \in \{1, 2, 3\}$ **do**
16 $e^j \leftarrow [e]_1^j + [e]_2^j$;
17 $f^j \leftarrow [f]_1^j + [f]_2^j$;
18 $\widehat{e^j} \leftarrow \widehat{[e]_1^j} + [e]_2^j$;
19 $\widehat{f^j} \leftarrow \widehat{[f]_1^j} + [f]_2^j$;
20 $e, f \leftarrow \min_{e^j, f^j} \{\mathit{dist}(e^j, \widehat{e^k}) + \mathit{dist}(f^j, \widehat{f^k}) \mid \mathit{flag}_j \wedge \widehat{\mathit{flag}}_j \wedge \mathit{flag}_k \wedge \widehat{\mathit{flag}}_k \wedge j \neq k\}_{j,k \in \{1,2,3\}}$;
21 $\widehat{[z]_1^{i_1}} \leftarrow \widehat{[c]_1^{i_1}} + e \cdot \widehat{[b]_1^{i_1}} + \widehat{[a]_1^{i_1}} \cdot f$;
22 $[z]_1^{i_2} \leftarrow [c]_1^{i_2} + e \cdot [b]_1^{i_2} + [a]_1^{i_2} \cdot f$;
23 $[z]_2^{i_3} \leftarrow [c]_2^{i_3} + e \cdot [b]_2^{i_3} + [a]_2^{i_3} \cdot f + e \cdot f$;
24 $[\mathbf{z}]_i \leftarrow ([z]_1^{i_1}, \widehat{[z]_1^{i_2}}, [z]_2^{i_3})$;
25 **return** $[\mathbf{z}]_i$;

C. Deep Learning in TrustDDL

In the context of deep learning, we typically encounter the following types of operations:

- *Linear Operations*: These operations involve basic arithmetic operations such as addition and multiplication.
- *Non-Linear Element-Wise Functions*: During model training and inference, activation functions such as *ReLU* and *Softmax* are used to introduce non-linearity to the output of a linear operation.
- *Local Transformations*: These operations involve modifying the shape of an input data structure to make it compatible with a specific layer or component of a deep learning model.

ASS-based protocols provide an efficient and straightforward means of implementing linear operations. Furthermore, each computing party can easily perform local transformations on its secret shares within TrustDDL. Additionally, TrustDDL is able to efficiently compute the *ReLU* activation function $f(x_i) = \max(0, x_i)$ using the *SecComp-BT* protocol. However, the *Softmax* activation function $f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$ cannot be computed with ASS-based protocols in TrustDDL. Consequently, TrustDDL delegates the computation of this activation function to the model owner during both model training and inference. As the *Softmax* activation function is typically applied only in the last layer of a neural network, over a relatively small number of neurons, even a resource-constrained computing party can usually handle this computation. TrustDDL also requires the model owner to compute the differentiation of the *Softmax* activation function.

IV. Performance Evaluation

A. Experimental Setup

We implemented TrustDDL in Python using the *Ray* framework for inter-party communication and *PyTorch* for data representation. Experiments ran on four machines, each featuring an Intel Xeon Gold 5120 CPU and 256 GB of RAM.

Table I details the neural network structure used in our experiments, including the utilized layers and their inputs/outputs. We evaluated test image accuracy over five training epochs with 60,000 images each. Fully connected layer weights were randomly initialized with a normal distribution $\mathcal{N}(0, 1/n)$, where n is the number of input neurons, and convolutional layer weights were initialized with a normal distribution $\mathcal{N}(0, 1/(k_1 \cdot k_2))$, where (k_1, k_2) denotes the kernel size. MNIST dataset feature values were normalized to $[0, 1]$. Furthermore, we assessed runtime and communication

Algorithm 5: *SecComp-BT* $([\mathbf{x}]_i, [\mathbf{y}]_i, [\mathbf{t}]_i, [\mathbf{B}]_i, i)$

Input: Shares $[\mathbf{x}]_i = ([x]_1^{i_1}, \widehat{[x]_1^{i_2}}, [x]_2^{i_3})$ and $[\mathbf{y}]_i = ([y]_1^{i_1}, \widehat{[y]_1^{i_2}}, [y]_2^{i_3})$ of the matrices to compare element-wise (i_1, i_2, and i_3 denote the sets of which party P_i holds shares), shares of an auxiliary matrix of positive numbers $[\mathbf{t}]_i = ([t]_1^{i_1}, \widehat{[t]_1^{i_2}}, [t]_2^{i_3})$, shares of a Beaver triple $[\mathbf{B}]_i = ([\mathbf{a}]_i, [\mathbf{b}]_i, [\mathbf{c}]_i)$ where $[\mathbf{a}]_i = ([a]_1^{i_1}, \widehat{[a]_1^{i_2}}, [a]_2^{i_3})$, $[\mathbf{b}]_i = ([b]_1^{i_1}, \widehat{[b]_1^{i_2}}, [b]_2^{i_3})$ and $[\mathbf{c}]_i = ([c]_1^{i_1}, \widehat{[c]_1^{i_2}}, [c]_2^{i_3})$, and the number i of the party calling this protocol;
Output: The sign of $x - y$, $sign(x - y)$;

$[\boldsymbol{\alpha}]_i \leftarrow [\mathbf{x}]_i - [\mathbf{y}]_i$;
$[\boldsymbol{\beta}]_i \leftarrow \mathit{SecMul}([\mathbf{t}]_i, [\boldsymbol{\alpha}]_i, [\mathbf{B}]_i, i)$;
for $i, j \in \{1, 2, 3\}$, $i \neq j$ **do**
  Send $\mathit{hash}([\boldsymbol{\beta}]_i)$ to P_j;
  Receive $\mathit{hash}([\boldsymbol{\beta}]_j)$ from P_j;
  $\mathit{hash}_{\beta,j} \leftarrow \mathit{hash}([\boldsymbol{\beta}]_j)$;
Confirm receipt of all the commitment values from all other parties;
for $i, j \in \{1, 2, 3\}$, $i \neq j$ **do**
  Send $[\boldsymbol{\beta}]_i$ to P_j;
  Receive and store $[\boldsymbol{\beta}]_j = ([\beta]_1^{j_1}, \widehat{[\beta]_1^{j_2}}, [\beta]_2^{j_3})$ from P_j;
  $\mathit{commit\_check}_j \leftarrow \mathit{hash}_{\beta,j} == \mathit{hash}([\boldsymbol{\beta}]_j)$;
  $\mathit{initializeFlags}(\mathit{flag}_{j_1}, \widehat{\mathit{flag}_{j_2}}, \mathit{flag}_{j_3}, \widehat{\mathit{flag}_{j_3}})$;
  $\mathit{flag}_{j_1}, \widehat{\mathit{flag}_{j_2}}, \mathit{flag}_{j_3}, \widehat{\mathit{flag}_{j_3}} \overset{\wedge}{\leftarrow} \mathit{commit\_check}_j$;
for $j \in \{1, 2, 3\}$ **do**
  $\beta^j \leftarrow [\beta]_1^j + [\beta]_2^j$;
  $\widehat{\beta^j} \leftarrow [\beta]_1^j + [\beta]_2^j$;
$\beta \leftarrow \min_{\beta^j} \{\mathit{dist}(\beta^j, \widehat{\beta^k}) \mid \mathit{flag}_j \wedge \widehat{\mathit{flag}_j} \wedge \mathit{flag}_k \wedge \widehat{\mathit{flag}_k} \wedge j \neq k\}_{j,k \in \{1,2,3\}}$;
$\mathit{sign}(x - y) \leftarrow \mathit{sign}(\beta)$;
return $\mathit{sign}(x - y)$;

TABLE I: Neural Network Configuration for MNIST Dataset

Input: 28×28 image
Convolution: $(28 \times 28) \rightarrow (14 \times 14 \times 5)$ kernel size (5×5), padding size 2, 5 output channels
ReLU: $(980) \rightarrow (980)$
FullyConnected: $(980) \rightarrow (100)$
ReLU: $(100) \rightarrow (100)$
FullyConnected: $(100) \rightarrow (10)$
Softmax: $(10) \rightarrow (10)$

costs via microbenchmarks, focusing on single-image (batch size 1) training or testing.

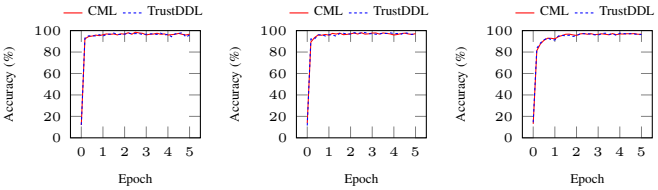
We used 64-bit fixed-point integers with 32 precision bits to convert floating-point values. During the *commitment* phase, TrustDDL utilized the SHA-256 hash function to hash shares.

B. Model Accuracy

Fig. 2 depicts the accuracy trained with CML (Centralized plaintext Model Learning) and TrustDDL. TrustDDL shows accuracy comparable to CML. This can be attributed to the utilization of the *SecComp-BT* protocol for computing the *ReLU* activation function, the outsourcing of the computation of the *Softmax* activation function to the model owner, and the utilization of 64-bit fixed-point integers with 20 precision bits to minimize accuracy loss during conversion between floating-point and fixed-point values during model training.

C. Runtime and Communication Cost

We compared TrustDDL with SecureNN [32], Falcon [33], and SafeML [34], all based on the additive secret-sharing scheme, in terms of runtime and communication costs. SecureNN provides security against *honest-but-curious* adversaries, while Falcon can detect and abort *malicious* behavior but cannot continue learning or inference. SafeML offers protection against *crash faults*.



(a) Batch Size 1, $\eta = 0.01$ (b) Batch Size 10, $\eta = 0.05$ (c) Batch Size 100, $\eta = 0.1$

Fig. 2: Model Accuracy on the MNIST Dataset

TABLE II: Runtime and Communication Cost

Framework	Model	Task	Time (s)	Comm. (MB)
SecureNN	Honest-but-Curious	Training	0.0824	6.8434
Falcon	Honest-but-Curious	Training	0.0513	1.7788
Falcon	Malicious	Training	0.0833	5.0660
SafeML	Crash-Fault	Training	6.3208	53.1408
TrustDDL	Honest-but-Curious	Training	5.9397	56.0285
TrustDDL	Malicious	Training	8.5315	68.4792
SecureNN	Honest-but-Curious	Inference	0.0498	4.0489
Falcon	Honest-but-Curious	Inference	0.0168	0.1535
Falcon	Malicious	Inference	0.0366	1.5624
SafeML	Crash-Fault	Inference	5.5406	28.0132
TrustDDL	Honest-but-Curious	Inference	5.1423	28.0132
TrustDDL	Malicious	Inference	7.6022	34.2384

In Table II, TrustDDL demonstrates higher runtime and communication costs compared to other frameworks due to its advanced capabilities in detecting and recovering from *Byzantine* faults. There is a notable increase in both runtime and communication costs when dealing with *malicious* and *crash-fault* models compared to *honest-but-curious* adversaries. However, TrustDDL shows a relatively lower escalation in costs from *honest-but-curious* to *malicious* adversaries. For instance, while Falcon exhibits a $0.62\times$ runtime increase, TrustDDL demonstrates a more modest $0.44\times$ increase.

V. Conclusion and Future Works

TrustDDL addresses privacy and Byzantine robustness concerns for model training and inference using a 3-party setting. Theoretically, it is secure against both *honest-but-curious* and *malicious* adversaries with an honest majority. It stands as a secret-sharing-based solution capable of detecting *Byzantine* failures and continuing protocol execution without interruption, ensuring reliable output delivery for deep learning tasks. Experimentally, TrustDDL demonstrates its ability for model training with satisfactory accuracy.

Future research for TrustDDL includes optimizing communication by designing protocols that reduce redundancy and improve efficiency in model training and inference.

Acknowledgment

This work was partially funded by the BMWK project SafeFBDC (01MK21002K) and the National Research Center ATHENE. We also want to thank hessian.AI at TU Darmstadt as well as DFKI Darmstadt for the support.

REFERENCES

[1] J. Duan, J. Zhou, and Y. Li, "Privacy-preserving distributed deep learning based on secret sharing," *Information Sciences*, vol. 527, pp. 108–127, Apr. 2020.

[2] J. Duan, J. Zhou, Y. Li, and C. Huang, "Privacy-preserving and verifiable deep learning inference based on secret sharing," *Neurocomputing*, vol. 483, pp. 221–234, Apr. 2022.

[3] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys*, vol. 52, no. 4, pp. 1–43, May 2019.

[4] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems*, Lake Tahoe, NV, USA, 2012, p. 1223–1231.

[5] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, Vienna, AustriaA, 2018, pp. 620–629.

[6] Y. Zou, X. Jin, Y. Li, Z. Guo, E. Wang, and B. Xiao, "Mariana: Tencent deep learning platform and its applications," in *Proceedings of the VLDB Endowment*, 2014, pp. 1772–1777.

[7] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an efficient and scalable deep learning training system," in *Proceedings of the 11th USENIX symposium on operating systems design and implementation*, Broomfield, CO, USA, 2014, pp. 571–582.

[8] H. C. Tanuwidjaja, R. Choi, S. Baek, and K. Kim, "Privacy-preserving deep learning on machine learning as a service—a comprehensive survey," *IEEE Access*, vol. 8, pp. 167 425–167 447, Sep. 2020.

[9] A. Boulemtafes, A. Derhab, and Y. Challal, "A review of privacy-preserving techniques for deep learning," *Neurocomputing*, vol. 384, pp. 21–45, Apr. 2020.

[10] Q. Xia, Z. Tao, and Q. Li, "Defenses against byzantine attacks in distributed deep neural networks," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 3, pp. 2025–2035, Jul. 2021.

[11] R. Guerraoui, N. Gupta, and R. Pinot, "Byzantine machine learning: A primer," *ACM Computing Surveys*, Aug. 2023. [Online]. Available: https://doi.org/10.1145/3616537

[12] B. Biggio, B. Nelson, and P. Laskov, "Poisoning attacks against support vector machines," in *Proceedings of the 29th International Coference on International Conference on Machine Learning*, Edinburgh, Scotland, 2012, p. 1467–1474.

[13] S. Mahloujifar, M. Mahmoody, and A. Mohammed, "Data poisoning attacks in multi-party learning," in *Proceedings of the 29th International Coference on International Conference on Machine Learning*, Long Beach, CL, USA, 2019, pp. 4274–4283.

[14] M. Fang, X. Cao, J. Jia, and N. Gong, "Local model poisoning attacks to Byzantine-Robust federated learning," in *Proceedings of the 29th USENIX security symposium*, Boston, MA, USA, 2020, pp. 1605–1622.

[15] Z. Ghodsi, T. Gu, and S. Garg, "SafetyNets: Verifiable execution of deep neural networks on an untrusted cloud," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, Long Beach, CA, USA, 2017, p. 4675–4684.

[16] S. Goldwasser, G. N. Rothblum, J. Shafer, and A. Yehudayoff, "Interactive proofs for verifying machine learning," in *Proceedings of 12th Innovations in Theoretical Computer Science Conference*, Virtual Event, 2021, pp. 41:1–41:19.

[17] C. Niu, F. Wu, S. Tang, S. Ma, and G. Chen, "Toward verifiable and privacy preserving machine learning prediction," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1703–1721, May 2020.

[18] A. Hassan, R. Hamza, H. Yan, and P. Li, "An efficient outsourced privacy preserving machine learning scheme with public verifiability," *IEEE Access*, vol. 7, pp. 146 322–146 330, Oct. 2019.

[19] Y. Fan, B. Xu, L. Zhang, J. Song, A. Zomaya, and K.-C. Li, "Validating the integrity of convolutional neural network predictions based on zero-knowledge proof," *Information Sciences*, vol. 625, pp. 125–140, May 2023.

[20] C. Weng, K. Yang, X. Xie, J. Katz, and X. Wang, "Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning," in *Proceedings of the 30th USENIX Security Symposium*, Virtual Event, 2021, pp. 501–518.

[21] M. Schwarz, S. Weiser, and D. Gruss, "Practical enclave malware with intel sgx," in *Proceedings of the 16th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, Gothenburg, Sweden, 2019, p. 177–196.

[22] S. Fei, Z. Yan, W. Ding, and H. Xie, "Security vulnerabilities of SGX and countermeasures: A survey," *ACM Computing Surveys*, vol. 54, no. 6, pp. 1–36, Jan. 2021.

[23] Z. Zhou, Q. Fu, Q. Wei, and Q. Li, "LEGO: A hybrid toolkit for efficient 2pc-based privacy-preserving machine learning," *Computers & Security*, vol. 120, p. 102782, Jun. 2022.

[24] F. Zheng, C. Chen, X. Zheng, and M. Zhu, "Towards secure and practical machine learning via secret sharing and random permutation," *Knowledge-Based Systems*, vol. 245, p. 108609, Apr. 2022.

[25] D. Alistarh, Z. Allen-Zhu, and J. Li, "Byzantine stochastic gradient descent," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, Montreal, Canada, 2018, p. 4618–4628.

[26] P. Blanchard, E. M. E. Mhamdi, R. Guerraoui, and J. Stainer, "Machine learning with adversaries: Byzantine tolerant gradient descent," in *Proceedings of the 31s International Conference on Neural Information Processing Systems*, Long Beach, CA, USA, 2017, p. 118–128.

[27] H. Guo, H. Wang, T. Song, Y. Hua, Z. Lv, X. Jin, Z. Xue, R. Ma, and H. Guan, "SIREN: Byzantine-robust federated learning via proactive alarming," in *Proceedings of the ACM Symposium on Cloud Computing*, Seattle, WA, USA, 2021, p. 47–60.

[28] T. D. Nguyen, P. Rieger, H. Chen, H. Yalame, H. Möllering, H. Fereidooni, S. M. , M. Miettinen, A. Mirhoseini, S. Zeitouni, F. Koushanfar, A. R. Sadeghi, and T. Schneider, "FLAME: Taming backdoors in federated learning," in *Proceedings of the 31st USENIX Security Symposium*, Boston, MA, USA, 2022, pp. 1415–1432.

[29] D. Beaver, "Efficient multiparty protocols using circuit randomization," in *Proceedings of Annual International Cryptology Conference (CRYPTO'91)*, Santa Barbara, CA, USA, 1991, pp. 420–432.

[30] L. Xiong, W. Zhou, Z. Xia, Q. Gu, and J. Weng, "Efficient privacy-preserving computation based on additive secret sharing," *arXiv preprint arXiv:2009.05356*, 2020.

[31] Z. Xia, Q. Gu, W. Zhou, L. Xiong, J. Weng, and N. Xiong, "STR: Secure computation on additive shares using the share-transform-reveal strategy," *IEEE Transactions on Computers*, vol. 73, no. 2, pp. 340–352, Apr. 2021.

[32] S. Wagh, D. Gupta, and N. Chandran, "SecureNN: 3-party secure computation for neural network training," in *Proceedings on Privacy Enhancing Technologies*, Stockholm, Sweden, 2019, p. 26–49.

[33] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin, "Falcon: Honest-majority maliciously secure framework for private deep learning," *arXiv preprint arXiv:2004.02229*, 2020.

[34] M. Mirabi, R. K. Nikiel, and C. Binnig, "SafeML: A privacy-preserving byzantine-robust framework for distributed machine learning training," in *Proceedings of the 2023 IEEE International Conference on Data Mining Workshops (ICDMW)*, Shanghai, China, 2023, pp. 207–216.

[35] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science*, Newport Beach, CA, USA, 2001, pp. 136–145.

VI. APPENDIX

The security of ASS-based protocols under the *honest-but-curious* adversary model is proven using the universal composability framework [35]. We assume their security, as established in prior works [30], [31], focusing solely on demonstrating TrustDDL's security during model training and inference. In the *honest-but-curious* adversary setting, computing parties in the proxy layer are hosted across diverse cloud infrastructures, each managed by a different cloud service provider. While assuming most providers avoid collusion due to conflicting interests, we consider the possibility of one computing party acting as the adversary (denoted as $\mathcal{A}$). The security of TrustDDL in this setting is established by

proving that the ideal experiment, where a simulator (denoted as $\mathcal{S}$) emulates $\mathcal{A}$'s view according to the functionality $\mathcal{F}$, is indistinguishable from the real experiment. The functionality $\mathcal{F}$ is defined so that a trusted functionality machine has access to the correct input information required for all operations related to learning and inference of a deep learning model. The ideal experiment aims to present $\mathcal{S}$ with computation results, ensuring that $\mathcal{S}$'s view is indistinguishable from the real-world view and that sensitive information remains concealed.

Theorem 6.1: TrustDDL is secure in the *honest-but-curious* adversary model.

Proof 6.1: During model training and inference, the view of the adversary can be defined as $view = ([s]_1^1, \widehat{[s]_1^2}, [s]_2^3, [s]_1^2, \widehat{[s]_1^3}, [s]_2^1, [s]_1^3, \widehat{[s]_1^1}, [s]_2^2)$, where these shares are masked computation results (e.g., the shares of e and f in the *SecMul-BT* protocol or the shares of β in the *SecComp-BT* protocol) from its own or received from the other two computing parties in the proxy layer. While the adversary $\mathcal{A}$ can reconstruct the secret of s using one of the following equations:

$$s = [s]_1^1 + [s]_2^1, \quad s = [s]_1^2 + [s]_2^2, \quad s = [s]_1^3 + [s]_2^3,$$
$$s = \widehat{[s]_1^1} + [s]_2^1, \quad s = \widehat{[s]_1^2} + [s]_2^2, \quad s = \widehat{[s]_1^3} + [s]_2^3$$

This secret is a masked value (e.g., the values e and f in the *SecMul-BT* protocol or the value of β in the *SecComp-BT* protocol). Therefore, the value of s is completely random and simulatable. Consequently, both the view of the adversary $\mathcal{A}$ and the outputs of the ASS-based protocols in TrustDDL are simulatable by the simulator $\mathcal{S}$, and the views of $\mathcal{S}$ and $\mathcal{A}$ are computationally indistinguishable.□

In the *malicious* adversary setting, we maintain the assumption that the computing parties within TrustDDL's proxy layer are hosted across diverse cloud infrastructures, operating under the premise of honest behavior to safeguard their reputations. However, we postulate the possibility of one computing party deviating from honesty. This party, referred to as a *Byzantine* party, intentionally provides incorrect computation results and manipulates data. The security of TrustDDL in this setting is established by demonstrating that every honest computing party can still derive masked correct computation results (e.g., the values e and f in the *SecMul-BT* protocol or the value of β in the *SecComp-BT* protocol) in the presence of a *Byzantine* party. In cases where honest computing parties receive erroneous shares, they can effectively detect the *Byzantine* party and reconstruct every masked correct computation result from a set of six reconstructions.

Theorem 6.2: TrustDDL is secure in the *malicious* adversary model.

Proof 6.2: Let party P_2 be a *Byzantine* (malicious) party. Based on the proposed solution in TrustDDL to resilience against a *Byzantine* party in the proxy layer, any honest computing party (i.e., P_1 and P_3) can reconstruct the masked value s using one of the following equations:

$$s = [s]_1^1 + [s]_2^1, \quad s = [s]_1^2 + [s]_2^2, \quad s = [s]_1^3 + [s]_2^3,$$
$$s = \widehat{[s]_1^1} + [s]_2^1, \quad s = \widehat{[s]_1^2} + [s]_2^2, \quad s = \widehat{[s]_1^3} + [s]_2^3$$

Now, there are three cases as follows:

- *Case 1*: If party P_2 violates the *commitment* phase by sending hash values of shares but later exchanges them with different shares, the two parties P_1 and P_3 can detect this misbehavior by recalculating the hash values and verifying if they match the values received during the *commitment* phase.
- *Case 2*: If party P_2 only violates the *commitment* phase with a specific party (e.g., P_3), party P_1 assumes both parties P_2 and P_3 are honest, while party P_3 detects party P_2 as a *Byzantine* party. Nevertheless, this does not hinder correct reconstructions, as TrustDDL's computing parties independently detect and recover from misbehavior.
- *Case 3*: If P_2 adheres to the *commitment* phase but uses incorrect shares for the calculation of the hash values, and in the share exchange, the two parties P_1 and P_3 can detect this misbehavior and identify the correct reconstructions with a very high probability by determining the minimum distance between any pair of two reconstructions.□

The security of TrustDDL against *Byzantine* attacks is ensured by honest behavior among cloud service providers, driven by their reputation concerns, the cooperation of most computing parties, and the *commitment* phase implementation.