MD-ML: Super Fast Privacy-Preserving Machine Learning for Malicious Security with a Dishonest Majority

Boshi Yuan[1], Shixuan Yang[1], Yongxiang Zhang[1], Ning Ding[1,2,*], Dawu Gu[1,2,*], and Shi-Feng Sun[1,2]

[1]*Shanghai Jiao Tong University, China*
[2]*Shanghai Jiao Tong University (Wuxi) Blockchain Advanced Research Center*
{nemoyuan2008,yangshixuan,zhang-yx7,dingning,dwgu,shifeng.sun}@sjtu.edu.cn

Abstract

Privacy-preserving machine learning (PPML) enables the training and inference of models on private data, addressing security concerns in machine learning. PPML based on secure multi-party computation (MPC) has garnered significant attention from both the academic and industrial communities. Nevertheless, only a few PPML works provide malicious security with a dishonest majority. The state of the art by Damgård *et al.* (SP'19) fails to meet the demand for large models in practice, due to insufficient efficiency. In this work, we propose MD-ML, a framework for Maliciously secure Dishonest majority PPML, with a focus on boosting online efficiency.

MD-ML works for n parties, tolerating corruption of up to $n-1$ parties. We construct our novel protocols for PPML, including truncation, dot product, matrix multiplication, and comparison. The online communication of our dot product protocol is one *single* element per party, *independent* of input length. In addition, the online cost of our multiply-then-truncate protocol is *identical* to multiplication, which means truncation incurs *no* additional online cost. These features are achieved for the *first* time in the literature concerning maliciously secure dishonest majority PPML.

Benchmarking of MD-ML is conducted for SVM and NN including LeNet, AlexNet, and ResNet-18. For NN inference, compared to the state of the art (Damgård *et al.*, SP'19), we are about 3.4–11.0× (LAN) and 9.7–157.7× (WAN) faster in online execution time.

1 Introduction

Machine learning (ML) is increasingly being applied in a wide variety of application domains. For instance, Neural networks (NN) have been extensively used in applications such as image and speech recognition, natural language processing, and financial forecasting. Although being a powerful tool that has been used to revolutionize various fields, ML is a data-driven approach, requiring a significant amount of data to train the models. This inevitably raises privacy concerns with regard to the data being used. The vast amount of personal data required for ML training creates opportunities for misuse and mishandling of sensitive information. As such, privacy-preserving techniques should be employed to ensure the privacy of the data being used. Besides model training, another dilemma of ML applications exists in inference services, where a company that owns a model provides ML inference services to customers. The company does not want to give out the model directly while the customers consider their data as private information and do not want to reveal it to others. Towards this, secure multi-party computation (MPC) is used to realize privacy-preserving machine learning (PPML) [40], addressing the dilemma of privacy concerns in ML.

MPC [20, 53] allows a set of mutually distrusting parties $P_1,\ldots,P_n$ to jointly compute a function $y = f(x_1,\ldots,x_n)$, where P_i holds its private input x_i. On a high level, MPC has the following security properties: (1) privacy—no set of t corrupted parties can learn more information than the output of the computation, and (2) correctness—no set of t corrupted parties can cause incorrect output.

The security of MPC can be formulated by requiring that execution of the protocol can be simulated and proven to be equivalent to the execution by a trusted third party [4, 36]. The assumed capabilities of an adversary can then be used to classify the level of security. Two of the most important security metrics are: (1) whether the adversary is assumed to be *semi-honest* or *malicious*, and (2) the number of parties the adversary is assumed to corrupt, more concretely, *honest majority* if the adversary can only corrupt less than 50% of the parties, and *dishonest majority* if it can corrupt 50% or more of the parties. With respect to the first security metric, a semi-honest adversary is assumed to honestly follow the protocol but tries to learn secret values (by analyzing the transcript of messages that it received and its internal state), whereas a malicious adversary is allowed to perform arbitrarily during the execution (*e.g.*, sending incorrect values). Malicious security (*i.e.*, security against a malicious adver-

*Corresponding authors.

sary) is more desired, providing stronger security guarantees, With respect to the second metric, dishonest majority is more desired as it enables two-party computation, and provides stronger collusion-resistance guarantees in multi-party computation.

Semi-honest PPML protocols have been successful in several previous works [6, 7, 27, 30, 39, 45, 49, 50, 52] in which efficiency has been significantly improved since the initial work [40]. Maliciously secure honest majority PPML protocols have also been developed with some success [51], some achieving fairness [6, 46] or guaranteed output delivery (GOD) [12, 28] in this security model.

Nevertheless, in real-world scenarios, the parties can be malicious or even collude to break the privacy of the data and the correctness of the results. In such cases, considering semi-honest or maliciously secure honest majority models does not perfectly satisfy the security requirements. The promotion of *maliciously secure dishonest majority* security model will significantly enhance the deployability of PPML.

Despite the success and efficiency achieved in semi-honest and malicious honest majority PPML, only a few works have explored PPML protocols considering the strong *maliciously secure dishonest majority* security model. Damgård *et al.* [15] pioneered this field by designing new cryptographic primitives and demonstrating their applications in decision trees and support vector machines (SVM) inference.

However as shown by Dalskov *et al.* [11], there is still a huge performance gap between protocols in the maliciously secure dishonest majority model and other weaker models, with the former being 1–3 orders of magnitude slower. For instance, [11] shows that the maliciously secure dishonest majority protocol in [15] is 277 times slower than the honest majority in MobileNet V1 1.0_224 inference, due to high communication costs. This limits the applications to larger ML models such as convolutional neural networks (CNNs).

It is worth noting that although Dalskov *et al.* [11] also consider malicious security with a dishonest majority, the underlying protocols remain the same with [15]. The quantization technique they consider is out of the scope of this paper. To the best of our knowledge, no follow-up work has improved the efficiency of PPML in this security model.

This work—We propose MD-ML, a framework providing fast, **M**aliciously secure, **D**ishonest majority pp**ML**. We focus on minimizing online communication costs and boosting online efficiency.

1.1 Our Contributions

We summarize our contributions below. For ease of reference, the term "SPD$\mathbb{Z}_{2^k}$+" is adopted to denote the state of the art by Damgård *et al.* [15], since their protocols are based on the SPD$\mathbb{Z}_{2^k}$ protocol [9].

Efficient Protocols for PPML. We extend the circuit-dependent preprocessing technique of TurboSpeedz [3] to the ring $\mathbb{Z}_{2^k}$, achieving the online communication of 1 ring element per party for each multiplication gate. Moreover, we construct our novel protocols for PPML, including (*cf.* Table 1):

Table 1: Comparison of theoretical online communication cost with SPD$\mathbb{Z}_{2^k}$+ [15]

Online comm. of	Ours	SPD$\mathbb{Z}_{2^k}$+
Multiplication	1	2
Vector dot product†	1	$2m$
Matrix multiplication‡	hw	$2hmw$
Matrix multiplication with truncation‡	hw in 1 round	$hw(2m+1)$ in 2 rounds

† Vector length is m.
‡ Sizes of matrices are $h \times m$ and $m \times w$.

- Multiplication with truncation (§4.1): Truncation is needed after each multiplication in fixed-point arithmetic (FPA), which is crucial for applications in ML. We integrate truncation operation with multiplication *without* any additional online communication cost. To the best of our knowledge, this feature is achieved for the *first* time in maliciously secure dishonest majority model.

- Vector dot product (§4.2): For vector dot product, we achieve the online communication of only one *single* element per party, *independent* of input vector length. To the best of our knowledge, this feature is achieved for the *first* time in maliciously secure dishonest majority security model.

- Matrix multiplication (§4.2): For multiplication of matrices with sizes $h \times m$ and $m \times w$, online communication cost is hw ring elements per party. This is $2m\times$ improvement over SPD$\mathbb{Z}_{2^k}$+ [15] which requires $2hmw$ elements online communication for each party.

- Secure Comparison (§4.3): To enable computing non-linear functions of ML models, we refine the existing mask-and-compare technique [15, 38] in the context of circuit-dependent preprocessing, and propose our efficient secure comparison procedure.

UC-secure PPML. Our framework works for any n parties, allowing malicious corruption of any $n-1$ parties. We rigorously show the security of all of our protocols in the UC framework [4], for the first time in maliciously secure dishonest majority PPML literature.

Implementing and Benchmarking. We implement our protocols and building blocks to show the practicality of our constructions in PPML. We carry out extensive benchmarks

on SVMs and NNs, in particular CNN models such as ResNet-18. The results show significant efficiency improvement of our constructions over the state of the art. For CNN inference, we are about 3.4–11.0× (LAN) and 9.7–157.7× (WAN) faster in online execution time compared to SPD$\mathbb{Z}_{2^k}$+ [15]. The highlights are summarized in Table 2.

Table 2: Improvement over SPD$\mathbb{Z}_{2^k}$+ [15] in terms of online execution time and communication for ML model inference

Model	Time		Comm.
	LAN	WAN	
SVM	4.3–12.8×	3.4–13.6×	1.9–2.1×
LeNet	3.4–9.2×	9.7–31.7×	2.0–2.9×
AlexNet	8.0–11.0×	93.3–157.7×	9.8–23.8×
Resnet-18†	25.8 s	362.9 s	4.15 GB

† ResNet-18 inference are not carried out for SPD$\mathbb{Z}_{2^k}$+, we instead provide performance of our MD-ML framework here.

1.2 Technical Overview

Throughout the paper, our protocols are divided into an input-independent preprocessing phase and an online phase, similar to many other works [9, 14]. We use the circuit-dependent preprocessing technique to improve efficiency. The main idea is to introduce some extra random values (in addition to multiplication triples [2]) in the preprocessing phase. Then the opening of some values, which would have to be done round-by-round in the online phase, can be moved to and batched in one round in the preprocessing phase.

We start from the TurboSpeedz [3] protocol, which introduces circuit-dependent preprocessing over a finite field $\mathbb{F}$. We extend the circuit-dependent preprocessing technique to the ring $\mathbb{Z}_{2^k}$. The motivation is that (1) when implementing on modern CPUs, computations over the ring $\mathbb{Z}_{2^k}$ runs faster [15], and (2) protocols PPML-specific operations (*e.g.*, truncation, comparison) are easier to design and implement over the ring $\mathbb{Z}_{2^k}$.

Then we design our novel protocols for PPML. While TurboSpeedz [3] is restricted to only arithmetic circuits (*i.e.*, only considers addition and multiplication of integers), we show that by using circuit-dependent preprocessing technique the online efficiency of PPML can be significantly improved.

To extend circuit-dependent preprocessing to the ring $\mathbb{Z}_{2^k}$, notice that TurboSpeedz [3] uses the SPDZ-sharing [13, 14] as the underlying secret-sharing scheme. It turns out that replacing it with the SPD$\mathbb{Z}_{2^k}$-sharing suffices. We further introduce some optimizations to reduce online computation and storage. We also analyze the security of the protocols to prove them UC-secure.

Then, to construct our novel protocols for PPML, note that for every secret value $[x]$, the parties obtain a secret-shared value $[\lambda_x]$ from the circuit-dependent preprocessing phase, and open $\Delta_x = \lambda_x + x$ in the online phase.

- For truncation, we integrate the truncation operation into multiplication. We observe that the $[\lambda]$-values can be used both in the opening of Δ-values and in the truncation pairs. We make the online cost of multiplication-then-truncation operations identical to that of multiplications.
- For vector dot product, instead of executing the multiplication procedure for each underlying element multiplications, we sample only one random $[\lambda]$-value for the result, and require communicating only one ring element, regardless of vector length.
- For comparison of secret values, we adopt the mask-and-compare approach [15, 38]. By circuit-dependent preprocessing, we push the opening of the masked value to the preprocessing phase to reduce communication rounds.

In order to establish the UC-security of our protocols, we integrate all of the aforementioned procedures into a preprocessing protocol Π_{PrepPPML} and an online protocol $\Pi_{\mathsf{OnlinePPML}}$. By defining the corresponding functionalities as $\mathcal{F}_{\mathsf{PrepPPML}}$ and $\mathcal{F}_{\mathsf{OnlinePPML}}$, we proceed to conduct the security proofs.

Finally, it is worth noting that several recent works [18, 45] also uses circuit-dependent preprocessing. In particular, ABY2.0 [45] also achieves constant online communication for dot product. We briefly discuss these works and compare them with our constructions in §3.3.

1.3 Other Related Work

MPC-based PPML. SecureML by Mohassel *et al.* [40] is the first to consider MPC-based PPML. The protocol makes use of the ABY framework [16] and is capable of training and inference on small NNs. ABY3 by Mohassel *et al.* [39] considers the 3-party case for both semi-honest and malicious security. ASTRA by Chaudhari *et al.* [6] improves this by proposing faster online protocols for malicious security. BLAZE by Patra *et al.* [46] proposes maliciously secure protocols that support inference of neural networks with fairness security. SWIFT by Koti *et al.* [28] achieves GOD security for both 3PC and 4PC PPML, and Tetrad [30] further improves efficiency for 4PC with the same security. Falcon by Wagh *et al.* [51] shows the feasibility of training large CNNs in malicious security settings. CryptGPU by Tan *et al.* [49] and Piranha by Watson *et al.* [52] implements MPC-based PPML on GPU, further enhancing efficiency. Dalskov *et al.* [11] studies secure evaluation of quantized neural networks (QNN). Keller *et al.* [26] extends the MP-SPDZ framework [23] to

implement faster secure deep learning training. For a more comprehensive survey of PPML, we refer the readers to [41].

The SPDZ-line of work. A major pillar in maliciously secure dishonest majority MPC is the SPDZ-line of work. The SPDZ protocol by Damgård *et al.* [13, 14] introduces an innovative secret-sharing scheme based on message authentication codes (MACs) and a series of protocols to enable MPC over a finite field $\mathbb{F}$. MASCOT by Keller *et al.* [24] improves the offline phase of SPDZ by replacing the expensive homomorphic encryption and zero-knowledge proofs with oblivious transfer. Overdrive by Keller *et al.* [25] further improves the efficiency of the offline phase by using global zero-knowledge proofs and BGV encryption. Concurrently with Overdrive, the SPD$\mathbb{Z}_{2^k}$ protocol by Cramer *et al.* [9] overcomes the hurdle of only being able to do maliciously secure dishonest majority MPC over a finite field. Overdrive2k by Orsini *et al.* [44] extends Overdrive to the ring $\mathbb{Z}_{2^k}$ by introducing a special packing technique for BGV encryption. Escudero *et al.* [19] achieve a lower amortized communication complexity for MPC over $\mathbb{Z}_{2^k}$ by exploiting Galois rings. For a more comprehensive survey of the SPDZ-line of work, we refer the readers to [43].

1.4 Organization of This Paper

In §2 we introduce notations, state the security model used in the paper, and give background on the SPD$\mathbb{Z}_{2^k}$ secret-sharing scheme. In §3 we give our protocols for arithmetic circuits over the ring $\mathbb{Z}_{2^k}$. In §4 we give efficient building blocks for PPML including multiplication with truncation, vector dot product, and comparison. We show how to build SVMs and NNs in §5. We benchmark our MD-ML framework and compare it with the state of the art in §6, and conclude in §7.

Appendices. Appendix A contains the functionalities. Appendix B contains the procedures. Appendix C contains the protocols.

2 Preliminaries

2.1 Notation

For an integer L, we denote by $\mathbb{Z}_L$ the set of integers $\{0,1,\ldots,L-1\}$, and use $\lambda \xleftarrow{\$} \mathbb{Z}_L$ to denote that λ is uniformly random in $\mathbb{Z}_L$. For a vector $\vec{x}$, denote by $\vec{x}[i]$ the i-th element of $\vec{x}$. The dot product of $\vec{x}$ and $\vec{y}$ is denoted by $\vec{x}\cdot\vec{y}$. For a k-bit integer λ, we explicitly use $(\lambda_0,\ldots,\lambda_{k-1})$ to denote the bit decomposition of λ where λ_0 is the least significant bit. This notation is employed to differentiate the individual bits of λ from other subscripts. Denote by $(x<y)$ the comparison result of x and y, *i.e.*, $(x<y)=1$ if $x<y$, $(x<y)=0$ otherwise. Table 3 summarizes the notation we use in this paper.

Table 3: Notation used throughout this paper

$P_1, P_2, \ldots, P_n$	Parties performing secure computation
$\mathbb{Z}_L$	The set of integers $\{0,1,\ldots,L-1\}$
$\lambda \xleftarrow{\$} \mathbb{Z}_L$	λ is uniformly random in $\mathbb{Z}_L$
$\vec{x}\cdot\vec{y}$	Dot product of vectors $\vec{x}$ and $\vec{y}$
$\vec{x}[i]$	The i-th element of vector $\vec{x}$
$(\lambda_0,\ldots,\lambda_{k-1})$	the bit decomposition of λ
$(x<y)$	Comparison result of $x<y$
$[x]$	SPD$\mathbb{Z}_{2^k}$ arithmetic share of x (§2.3)
$[x]_2$	SPD$\mathbb{Z}_{2^k}$ binary share of $x \in \mathbb{Z}_2$ (§2.3)
k, s	Concrete parameters for SPD$\mathbb{Z}_{2^k}$ (§2.3)

2.2 Security Model

Our protocols work with n parties, and we consider security against a malicious, static adversary corrupting up to $n-1$ parties. In this context, "malicious" denotes that the corrupted parties may deviate arbitrarily from the protocol (*e.g.*, send incorrect values), and "static" denotes that corruption may only take place before protocol starts and remains fixed throughout protocol execution.

We prove our security statements in the universal composition (UC) framework of Canetti [4], where security is argued by the indistinguishability of an ideal world, modeled by a *functionality* (denoted by $\mathcal{F}$ with some subscript), and the real world, instantiated by a *protocol* (denoted by Π with some subscript). Protocols can invoke *procedures* (denoted by π with some subscript), which are like protocols but not intended to instantiate a functionality. Instead they are used as subroutines inside other protocols that instantiates some functionality.

2.3 The SPD$\mathbb{Z}_{2^k}$ Secret-sharing Scheme

We briefly recall the SPD$\mathbb{Z}_{2^k}$ secret-sharing scheme [9] in this section, which enables n parties to compute over the ring $\mathbb{Z}_{2^k}$ for any k (*e.g.*, $k=64$ for arithmetic circuits and $k=1$ [15] for boolean circuits). The main idea is that the parties carry out computations over the ring $\mathbb{Z}_{2^{k+s}}$, where $\sigma = s - \log s$ is the statistical security parameter, but security and correctness is only guaranteed modulo 2^k.

Definition of SPD$\mathbb{Z}_{2^k}$ secret-sharing scheme $[\cdot]_{2^k}$. Assume each party P_i (for $i=1,2,\ldots,n$) holds a uniformly random additive share $\alpha^i \xleftarrow{\$} \mathbb{Z}_{2^s}$ of a secret global MAC key $\alpha = \sum_{i=1}^n \alpha^i \bmod 2^{k+s}$. An element $x \in \mathbb{Z}_{2^k}$ is $[\cdot]_{2^k}$-shared if each party holds $x^i \in \mathbb{Z}_{2^{k+s}}$ and $m_x^i \in \mathbb{Z}_{2^{k+s}}$, where $x = \sum_{i=1}^n x^i \bmod 2^k$ is the element being shared, and $m_x = \sum_{i=1}^n m_x^i \bmod 2^{k+s}$ is the MAC, such that $m_x = (\sum_{i=1}^n x^i)\cdot\alpha \bmod 2^{k+s}$. We denote $[x]_{2^k} = \big((x^1,\ldots,x^n),(m_x^1,\ldots,m_x^n)\big)$. We abbreviate $[x]_{2^k}$ as $[x]$ when 2^k is a large integer (*e.g.*, $k=64$) and the context is

clear.

Linear combination of $[\cdot]$-sharing. The linear combination of secret-shared values can be locally computed by the parties. For instance, given secret-shared values $[x] = \big((x^1,\ldots,x^n),(m_x^1,\ldots,m_x^n)\big)$, $[y] = \big((y^1,\ldots,y^n),(m_y^1,\ldots,m_y^n)\big)$ and public constants a,b,c, the linear combination $[z] = a[x] + b[y] + c$ can be computed as $[z] = \big((z^1,\ldots,z^n),(m_z^1,\ldots,m_z^n)\big)$ where P_i locally computes $z^i = \begin{cases} ax^1 + by^1 + c, & \text{if } i = 1 \\ ax^i + by^i, & \text{if } i \neq 1 \end{cases}$ and $m_z^i = am_x^i + bm_y^i + c\alpha^i$ for $i = 1,\ldots,n$.

Opening $[\cdot]$-shared values and checking the MAC. To open $[x]$ so that all parties learn x in the clear, each party P_i use a random shared value $[r]$ to mask the upper s bits of x, getting $[\tilde{x}] = [x] + 2^k[r]$. Denote P_i's share and MAC share on $[\tilde{x}]$ by $\tilde{x}^i$ and $m_{\tilde{x}}^i$. Each party P_i broadcasts its own $\tilde{x}^i$ (but not $m_{\tilde{x}}^i$) and, upon receiving the values from other parties, computes $x = \sum_{i=1}^n \tilde{x}^i \bmod 2^k$. To ensure correctness, the MAC $m_{\tilde{x}}$ should be checked, but can be deferred and batched with other opened values. Procedure π_{MACCheck} of [9] takes as input a set of opened values and verifies the MACs of all the values, with failure probability negligible in s. For completeness, π_{MACCheck} is provided in Appendix B.1.

3 Efficient Protocols for Arithmetic Circuits

In this section, we provide our protocols for computing arithmetic circuits (*i.e.*, additions and multiplications) over the ring $\mathbb{Z}_{2^k}$. We extend the circuit-dependent preprocessing technique of TurboSpeedz [3] to MPC over the ring $\mathbb{Z}_{2^k}$. In §3.1 we provide a concise overview of the TurboSpeedz protocol. In §3.2 we propose our modifications and optimizations, then formally present our protocols and state the security theorems. It is worth noting that several recent works [18, 45] also uses circuit-dependent preprocessing. Thus, in §3.3 we discuss the techniques of ABY2.0, and explain why these techniques fails in the security model we concern.

3.1 An Overview of TurboSpeedz

The TurboSpeedz protocol [3] uses the SPDZ-sharing (denoted by $\langle\cdot\rangle$) as its underlying secret-sharing scheme. The SPDZ-sharing is the same as the SPD$\mathbb{Z}_{2^k}$-sharing (§2.3, denoted by $[\cdot]$), except that the secret data, the MAC key, and the sharings are from a finite field $\mathbb{F}$, instead of the ring $\mathbb{Z}_{2^{k+s}}$.

The protocol is divided into a circuit-dependent preprocessing phase and an online phase. Denote by x,y,z, *etc.* the wires in the circuit. The parties are assumed to have the following circuit-dependent preprocessing material: for every wire z in the circuit, the parties have a shared value $\langle\lambda_z\rangle$. If z is not the output of an addition gate, then λ_z is a uniformly random value. If z is the output of an addition gate with input wires x,y, then λ_z is defined (recursively) as $\lambda_z = \lambda_x + \lambda_y$. In addition, for each multiplication gate parties are assumed to have a shared multiplication triple $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ and values δ_x, δ_y, where $c = ab$, $\delta_x = a - \lambda_x$, $\delta_y = b - \lambda_y$.

Then, in the online phase, for a secret value x on wire x[1], parties are assumed to hold $\langle x\rangle$ and are required to learn $\Delta_x = x + \lambda_x$ in the clear. For an addition gate with input (x,y), it is clear that the parties can compute $\langle z\rangle = \langle x\rangle + \langle y\rangle$ and $\Delta_z = \Delta_x + \Delta_y$ locally. For a multiplication gate with input x,y, to get the result $\langle z\rangle$, the parties can compute the output as $\langle z\rangle = (\Delta_x + \delta_x)(\Delta_y + \delta_y) - (\Delta_y + \delta_y)\langle a\rangle - (\Delta_x + \delta_x)\langle b\rangle + \langle c\rangle$. This is correct since $\Delta_x + \delta_x = (x + \lambda_x) + (a - \lambda_x) = x + a$, and similarly $\Delta_y + \delta_y = y + b$, hence for the right-hand-side of $\langle z\rangle$ we have: $(\Delta_x + \delta_x)(\Delta_y + \delta_y) - (\Delta_y + \delta_y)a - (\Delta_x + \delta_x)b + c = (x + a)(y + b) - (y + b)a - (x + a)b + ab = xy = z$. Finally, to retain the knowledge of Δ-values, parties compute $\langle\Delta_z\rangle = \langle z\rangle + \langle\lambda_z\rangle$ and open it to get Δ_z.

3.2 Our Protocols

For our protocols, to enable computation over the ring $\mathbb{Z}_{2^k}$, we find that changing the underlying secret-sharing scheme from SPDZ-sharing ($\langle\cdot\rangle$) to SPD$\mathbb{Z}_{2^k}$-sharing ($[\cdot]$) suffices. However, the security of the protocols needs to be re-analyzed, as will be done later in this section.

Further optimizations can be introduced. In the aforementioned procedure, for a multiplication gate with output wire z, parties have to compute $[z]$, followed by computing $[\Delta_z] = [z] + [\lambda_z]$ and opening Δ_z. Patra *et al.* have noted in ABY2.0 [45] that it is unnecessary for the parties to compute and store the value $[z]$. Instead, only $[\Delta_z]$ is needed. This also holds for addition gates, where parties only need to compute $\Delta_z = \Delta_x + \Delta_y$ locally, without computing and storing $[z]$. We adopt this optimization and save 2 ring elements of storage in the online phase per wire per party.

We note that it is imperative for the parties to retain the knowledge of the $[\lambda]$-values corresponding to each wire during the preprocessing phase. Subsequently, during the online phase, for each gate the parties must compute and open the Δ-values associated with the output wire of that gate, based on the Δ-values of the input wires. This requirement holds true for all of the protocols proposed in §3 and §4.

For the preprocessing protocol, we rely on the SPD$\mathbb{Z}_{2^k}$ preprocessing functionality $\mathcal{F}_{\mathsf{Prep}}$ [9] for generating randomness. $\mathcal{F}_{\mathsf{Prep}}$ has three commands: Triple for generating a multiplication triple, Rand for generating a random shared element for all the parties, and Input for generating a random shared element $[r]$, where r is known by exactly one party in the clear. The formal definition of $\mathcal{F}_{\mathsf{Prep}}$ is given in Appendix A.1.

We are now ready to present the arithmetic-circuit preprocessing protocol $\Pi_{\mathsf{PrepArith}}$. The goal of $\Pi_{\mathsf{PrepArith}}$ is to generate the following material for the circuit: (1) a shared $[\lambda]$-value for each wire, (2) a shared multiplication triple for each multiplication gate, and (3) two public δ-values for each

[1] Note that we do not distinguish between the wire and its value, like the SPDZ-line of work [9, 14, 24].

multiplication gate. The above features are captured by the functionality $\mathcal{F}_{\mathsf{PrepArith}}$, which is described in Appendix A.2 due to space constraints. The protocol $\Pi_{\mathsf{PrepArith}}$ is presented below.

Protocol 1: $\Pi_{\mathsf{PrepArith}}$

The parties proceed in topological order of the circuit.
Input: For each input wire x of party P_i, parties call $\mathcal{F}_{\mathsf{Prep}}$.Input to get $([\lambda_x], \lambda_x)$ where λ_x is known to P_i in the clear, and all parties hold the sharing $[\lambda_x]$.
Add: For an addition gate with input wires x, y and output wire z, all the parties locally compute $[\lambda_z] = [\lambda_x] + [\lambda_y]$.
Multiply: For a multiplication gate with input wires x, y and output wire z, parties do the following:

1. Call $\mathcal{F}_{\mathsf{Prep}}$.Triple to get a multiplication triple $([a], [b], [c])$.
2. Locally compute $[\delta_x] = [a] - [\lambda_x]$ and $[\delta_y] = [b] - [\lambda_y]$.
3. Call $\mathcal{F}_{\mathsf{Prep}}$.Rand to get $[\lambda_z]$, where $\lambda_z \xleftarrow{\$} \mathbb{Z}_{2^k}$.

Now the parties have $[a], [b], [c], [\lambda_z], [\delta_x], [\delta_y]$ for a multiplication gate.
Output: Parties open the shared $[\delta]$-values for every multiplication gate to get them in the clear, then run π_{MACCheck} to check the MACs on values that are opened. If π_{MACCheck} aborts then the parties abort. Otherwise the protocol completes without failure.

The goal of the online protocol $\Pi_{\mathsf{OnlineArith}}$ is to let parties interactively obtain the Δ-values for each wire in topological order, and obtain the final output on their secret inputs, using the preprocessed material obtained in $\Pi_{\mathsf{PrepArith}}$. We capture these features in functionality $\mathcal{F}_{\mathsf{OnlineArith}}$, which is described in Appendix A.2 due to space constraints. The protocol $\Pi_{\mathsf{OnlineArith}}$ is presented below.

Protocol 2: $\Pi_{\mathsf{OnlineArith}}$

Initialize: Parties call $\mathcal{F}_{\mathsf{PrepArith}}$ with the circuit to get the δ-values, the shared $[\lambda]$-values, and the multiplication triples for each gate.
Then, parties proceed in topological order.
Input: For P_i to share its input value x, P_i computes and broadcasts $\Delta_x = x + \lambda_x$, then all parties store Δ_x.
Add: For an addition gate with input wires x, y and output wire z, parties locally compute $\Delta_z = \Delta_x + \Delta_y$.
Multiply: For a multiplication gate with input wires x, y and output wire z, all parties do the following:

1. Locally compute $[\Delta_z] = (\Delta_x + \delta_x)(\Delta_y + \delta_y) - (\Delta_y + \delta_y)[a] - (\Delta_x + \delta_x)[b] + [c] + [\lambda_z]$.
2. Open Δ_z to get Δ_z in the clear.

Output: To output the value on wire x, all the parties do the following:

1. Locally compute $[x] = \Delta_x - [\lambda_x]$ and open x in the clear.
2. Run π_{MACCheck} on all the values that have been opened so far. If π_{MACCheck} aborts then parties abort. Otherwise the output x is correct.

Now we state the security of our protocols. The proofs of the following theorems are given in the full version.

Theorem 1. *The protocol $\Pi_{\mathsf{PrepArith}}$ securely implements the functionality $\mathcal{F}_{\mathsf{PrepArith}}$ against a static, malicious adversary corrupting up to $n-1$ parties in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model.*

Theorem 2. *The protocol $\Pi_{\mathsf{OnlineArith}}$ securely implements the functionality $\mathcal{F}_{\mathsf{OnlineArith}}$ against a static, malicious adversary corrupting up to $n-1$ parties in the $\mathcal{F}_{\mathsf{PrepArith}}$-hybrid model.*

3.3 Comparison with Other Works

3.3.1 Comparison with ABY2.0

ABY2.0 by Patra *et al.* [45] also uses the circuit-dependent preprocessing technique to evaluate arithmetic circuits in *semi-honest two-party* computation. It is the state-of-the-art mixed-circuit framework in this security model. ABY2.0 achieves less communication in the preprocessing phase and the protocols are more concise. Nevertheless, the techniques of ABY2.0 cannot be directly applied to maliciously secure dishonest majority MPC, as demonstrated below.

ABY2.0 works similar as us: for a multiplication gate with input wires x, y and output wire z, the parties are assumed to have the sharings of circuit-dependently preprocessed values $\lambda_x, \lambda_y, \lambda_z$. The definition of these λ-values are the same as ours. However, instead of letting the parties have a multiplication triple (a, b, c) (which are independent of the λ-values), ABY2.0 let the parties hold a sharing of λ_{xy}, where $\lambda_{xy} = \lambda_x \lambda_y$ (*i.e.*, $(\lambda_x, \lambda_y, \lambda_{xy})$ forms a multiplication triple). This is done by having the parties execute a semi-honest triple generation protocol on inputs λ_x, λ_y.

This does not work for maliciously secure dishonest majority MPC, even if the underlying secret-sharing scheme of ABY2.0 is replaced by the SPD$\mathbb{Z}_{2^k}$-sharing. The triple generation protocol in malicious setting [9] cannot take two determined values (λ_x, λ_y in the above case) and output their product, since it involves a random linear combination step that randomizes the input values to ensure privacy in the presence of a malicious adversary.[2] Hence, the techniques of ABY2.0 cannot be directly applied to our protocols.

[2] Although the product of two determined values *can* be computed using a random triple, this incurs high communication cost and round complexity.

3.3.2 Discussions on flexible corruption

In another line of work, Goyal *et al.* [21] uncover the benefits of allowing flexible corruption wherein the corruption threshold is $t = n(1-\varepsilon)$ for a constant $0 < \varepsilon < 1/2$, as opposed to the conventional dishonest majority threshold of $t = n-1$ (n is the number of parties). Unlike most MPC protocols (including this work) where global communication grows linearly with n, the (amortized) global communication in [21] remains bounded by a constant, regardless of the growth of n. Subsequently, Escudero *et al.* [18] reduce this constant for the online phase by using the circuit-dependent preprocessing technique.

It is natural to explore the applicability of these techniques to our protocols for improved performance. However, the aforementioned works rely on packed Shamir secret-sharing, which is currently limited to arithmetic circuits over a moderately large finite field. Adapting these techniques for PPML necessitates either (1) extending packed Shamir secret-sharing to the ring $\mathbb{Z}_{2^k}$, or (2) devising ML operations (truncation, comparison, *etc.*) for packed Shamir secret-sharing over a finite field $\mathbb{F}$. Several works have been dedicated to both avenues. Regarding (1), Abspoel *et al.* [1] and Cramer *et al.* [10] propose techniques to transform Shamir secret-sharing to $\mathbb{Z}_{2^k}$. As for (2), Liu *et al.* [37] propose new truncation and comparison protocols for Shamir secret-sharing over $\mathbb{F}$. Nevertheless, these works primarily consider *non*-packed Shamir secret-sharing, leaving uncertainty regarding the compatibility and concrete efficiency of these techniques for packed Shamir-secret sharing. This issue is left as an open problem.

4 Building Blocks for Machine Learning

In this section, we further exploit the circuit-dependent preprocessing technique, and develop our novel building blocks for privacy-preserving machine learning. These include multiplication with truncation (§4.1), vector dot product (§4.2), and secure comparison (§4.3). We show that all of these building blocks benefit from circuit-dependent preprocessing, resulting in less online communication cost. In §4.4 we analyze the security and discuss the reactiveness of our constructions.

4.1 Multiplication with Truncation

Most of the computations in machine learning involve operating over decimals. To represent decimal values, we make use of fixed-point arithmetic (FPA). Values are represented as signed two's complement over the ring $\mathbb{Z}_{2^k}$ [40], with the most significant bit being the sign bit, and d least significant bits being the fractional part. Note that the computations are still carried out in $\mathbb{Z}_{2^{k+s}}$, but we only care about the values of the lower k bits. In FPA representation, the number of fractional bits doubles after each multiplication, therefore the multiplication result needs to be divided by 2^d (*i.e.*, truncate the last d bits) to avoid overflowing the ring $\mathbb{Z}_{2^k}$.

We provide a brief overview of existing truncation techniques. In ABY3 [39], the protocol involves generating shared random values $([r'],[r])$ in the preprocessing phase, where $r = r'/2^d$ (referred to as a truncation pair). During the online phase, parties compute and open $z'+r'$, followed by local computation of $[z] = (z'+r')/2^d - [r]$ to obtain the desired result. This approach requires one round of communication for multiplication and an additional round for truncation.

Subsequent works such as ABY2.0 [45], Swift [28], Tetrad [30], and MPClan [29] optimize the online cost of truncation by integrating it into the multiplication process. Instead of opening the intermediate result after multiplication and opening the final result after truncation, the intermediate result is masked with the random $[r']$ from the truncation pair and opened. This integration of the opening of masked value into the multiplication procedure helps reduce communication costs.

Inspired by the aforementioned techniques, we observe that in the multiplication procedure of $\Pi_{\mathsf{OnlineArith}}$, where $\Delta_{z'} = z' + \lambda_{z'}$ is opened, it is possible to locally truncate it to obtain $\Delta_z = \Delta_{z'}/2^d$. Assuming the parties possess $[\lambda_z]$ such that $\lambda_z = \lambda_{z'}/2^d$, it follows that $z = \Delta_z - \lambda_z = \Delta_{z'}/2^d - \lambda_{z'}/2^d = z'/2^d$, thereby completing the truncation. The crucial point is that $([\lambda_{z'}],[\lambda_z])$ not only functions as random masks in $\Delta_{z'}$ and Δ_z, but also serves as a truncation pair.

In order to generate the pair $([\lambda_{z'}],[\lambda_z])$ in the preprocessing phase, we no longer rely on $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Rand}$ as in $\Pi_{\mathsf{PrepArith}}$. Instead, we utilize the functionality $\mathcal{F}_{\mathsf{edaBits}}$ for edaBits generation [17] to obtain the $(k-d)$-bit value $[\lambda_z]$ and a d-bit value $[u]$. Then $[\lambda_{z'}]$ can be locally computed as $[\lambda_{z'}] = 2^d \cdot [\lambda_z] + [u]$. The formal definition $\mathcal{F}_{\mathsf{edaBits}}$ is given in Appendix A.4. The procedure $\pi_{\mathsf{MultTrunc}}$ is described below.

Procedure 1: $\pi_{\mathsf{MultTrunc}}$

For a multiply-then-truncate gate with input wires x, y and output wire z, to multiply x, y and truncate the result by d bits:

Preprocessing phase:

1. Parties call $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Triple}$ to get $([a],[b],[c])$.
2. Parties compute $[\delta_x] = [a] - [\lambda_x]$, $[\delta_y] = [b] - [\lambda_y]$ and open δ_x, δ_y.
3. Parties call $\mathcal{F}_{\mathsf{edaBits}}$ on input $(k-d)$ to get $[\lambda_z]$ and its bit decomposition $\{[\lambda_{z,i}]_2\}_{i=0}^{k-d-1}$.
4. Parties call $\mathcal{F}_{\mathsf{edaBits}}$ on input d to get $[u]$ and its bit decomposition $\{[u_i]_2\}_{i=0}^{d-1}$.
5. Parties locally compute $[\lambda_{z'}] = 2^d \cdot [\lambda_z] + [u]$.

Online phase:

1. Parties locally compute $[\Delta_{z'}] = (\Delta_x + \delta_x)(\Delta_y + \delta_y) - (\Delta_y + \delta_y)[a] - (\Delta_x + \delta_x)[b] + [c] + [\lambda_{z'}]$.
2. Parties open $\Delta_{z'}$ to get $\Delta_{z'}$ in the clear.
3. Parties locally compute $\Delta_z = \Delta_{z'}/2^d$.

Online communication cost. In the online phase of procedure $\pi_{\mathsf{MultTrunc}}$, each party is only required to broadcast a single ring element. This communication cost is *identical* to that of multiplication in $\Pi_{\mathsf{OnlineArith}}$. Therefore, truncation is essentially performed at *no* additional online communication cost. To the best of our knowledge, this feature is achieved for the *first* time in maliciously secure dishonest majority PPML protocols.

Correctness. For the correctness of $\pi_{\mathsf{MultTrunc}}$ we have the following proposition.

Proposition 3. *In the procedure* $\pi_{\mathsf{MultTrunc}}$*, if* $xy \le 2^\ell$ *for some* $\ell < k$*, then with probability at least* $1 - 2^{\ell-k}$ *it holds that* $z = \lfloor xy/2^d \rfloor + v$ *for some* $v \in \{0,1\}$.

Proof. We use the notations in $\pi_{\mathsf{MultTrunc}}$ and let $z' = xy = \Delta_{z'} - \lambda_{z'}$. Denote by $\mathsf{Carry}_k(x,y)$ the k-th carry bit when adding x and y. Since $\Delta_{z'} = (\lambda_{z'} + z') \bmod 2^k$, it holds that $\Delta_{z'} = \lambda_{z'} + z' - 2^k u$ with $u = \mathsf{Carry}_k(\lambda_{z'}, z')$. Also, we have $(\Delta_{z'} \bmod 2^d) = (\lambda_{z'} \bmod 2^d) + (z' \bmod 2^d) - 2^d v$ with $v = \mathsf{Carry}_d(\lambda_{z'}, z')$. Since for all positive integers x it holds that $\lfloor x/q \rfloor = (x - (x \bmod q))/q$, we can see that z satisfies

$$
\begin{aligned}
z = \Delta_z - \lambda_z &= \lfloor \Delta_{z'}/2^d \rfloor - \lambda_z = \frac{\Delta_{z'} - (\Delta_{z'} \bmod 2^d)}{2^d} - \lambda_z \\
&= \frac{\lambda_{z'} + z' - 2^k u - (\lambda_{z'} \bmod 2^d) - (z' \bmod 2^d) + 2^d v}{2^d} - \lambda_z \\
&= \frac{\lambda_{z'} - (\lambda_{z'} \bmod 2^d)}{2^d} + \frac{z' - (z' \bmod 2^d)}{2^d} - 2^{k-d}u + v - \lambda_z \\
&= \lambda_z + \lfloor z'/2^d \rfloor - 2^{k-d}u + v - \lambda_z = \lfloor z'/2^d \rfloor - 2^{k-d}u + v,
\end{aligned}
$$

hence the error is $-2^{k-d}u + v$. Since $u = \mathsf{Carry}_k(\lambda_{z'}, z') = (\lambda_{z'} + z' \ge 2^k)$, the probability that $u = 1$ is the probability that $\lambda_{z'} \ge 2^k - z'$. Moreover, since $z' < 2^\ell$ and $\lambda_{z'}$ is uniformly random in $\mathbb{Z}_{2^k}$, this probability is upper bounded by $2^{\ell-k}$. □

4.2 Vector Dot Product

In this section, we show that the online communication of vector dot product can be made *independent* of the vector size. This has been achieved in either semi-honest setting [45] or malicious setting with an honest majority [8, 39]. To the best of our knowledge, no previous works have achieved similar results in maliciously secure dishonest majority model.

For convenience we allow the $[\cdot]$ notation for vectors, *i.e.*, $[\vec{\lambda}]$ indicates that each element of the vector $\vec{\lambda}$ is $[\cdot]$-shared.

For a dot product gate with input vectors $\vec{x}, \vec{y}$ of length m, the goal of the procedure $\pi_{\mathsf{DotProduct}}$ is to compute the output on wire z where $z = \vec{x} \cdot \vec{y} = \sum_{i=1}^m \vec{x}[i]\vec{y}[i]$. A trivial way is to execute m multiplications in $\Pi_{\mathsf{OnlineArith}}$ for each underlying multiplication, but this results in online communication linear in vector length m.

We now show how to make online communication independent of vector length. Observe that in preprocessing phase parties do not need to generate the random shares $[\overrightarrow{\lambda_z}[i]]$ for each multiplication in the dot product. Instead, only a single random share $[\lambda_z]$ is needed for the final result. Then in the online phase, parties locally compute and sum up the products before masking with $[\lambda_z]$ to get $[\Delta_z]$. Finally, parties open Δ_z to retain knowledge on Δ_z. For details, see our dot product procedure $\pi_{\mathsf{DotProduct}}$ below.

Procedure 2: $\pi_{\mathsf{DotProduct}}$

For a dot product gate with input vectors of length m on wires $\vec{x}, \vec{y}$:

Preprocessing phase:

1. Parties call $\mathcal{F}_{\mathsf{Prep}}$.Rand to get $[\lambda_z]$ where $\lambda_z \xleftarrow{\$} \mathbb{Z}_{2^k}$.
2. Parties call $\mathcal{F}_{\mathsf{Prep}}$.Triple m times to get m triples, obtaining $[\vec{a}], [\vec{b}], [\vec{c}]$, where $([\vec{a}[i]], [\vec{b}[i]], [\vec{c}[i]])$ is an individual multiplication triple obtained in the i-th call of $\mathcal{F}_{\mathsf{Prep}}$.Triple.
3. Parties compute $[\overrightarrow{\delta_x}] = [\vec{a}] - [\overrightarrow{\lambda_x}]$, $[\overrightarrow{\delta_y}] = [\vec{b}] - [\lambda_y]$ and open $\overrightarrow{\delta_x}, \overrightarrow{\delta_y}$.

Online phase:

1. Parties locally compute $[\Delta_z] = \sum_{i=1}^m \Big((\overrightarrow{\Delta_x}[i] + \overrightarrow{\delta_x}[i])(\overrightarrow{\Delta_y}[i] + \overrightarrow{\delta_y}[i]) - (\overrightarrow{\Delta_y}[i] + \overrightarrow{\delta_y}[i])[\vec{a}[i]] - (\overrightarrow{\Delta_x}[i] + \overrightarrow{\delta_x}[i])[\vec{b}[i]] + [\vec{c}[i]] \Big) + [\lambda_z]$.
2. Parties open Δ_z to get Δ_z in the clear.

Online communication cost. For length-m vector dot product, SPD$\mathbb{Z}_{2^k}$+ [15] requires communication of $2m$ elements for each party in the online phase, while we only require 1 *single* element for each party, which is *independent* of input vector length m. To the best of our knowledge, this feature is achieved for the *first* time in maliciously secure dishonest majority MPC.

Matrix multiplication. Since matrix multiplication consists of vector dot products, it benefits from our constructions. Concretely, to multiply a secret $h \times m$ matrix by a secret $m \times w$ matrix, SPD$\mathbb{Z}_{2^k}$+ [15] requires online communication of $2mhw$ ring elements per party, while we require only hw ring elements, which is $2m\times$ improvement.

Dot product and matrix multiplication with truncation.

The *dot-product-then-truncate* procedure $\pi_{\mathsf{DotProductTrunc}}$ can be constructed similarly as $\pi_{\mathsf{MultTrunc}}$. We describe the details of $\pi_{\mathsf{DotProductTrunc}}$ in Appendix B.2. In addition, matrix multiplication with truncation also benefits from $\pi_{\mathsf{DotProductTrunc}}$. The property that truncation requires no additional online communication cost still holds. Concretely, for length m vector dot product with truncation, our construction requires only 1 ring element of online communication per party in 1 round, whereas SPD$\mathbb{Z}_{2^k}$+ [15] requires $2m+1$ elements in 2 rounds. For size $h \times m$ and $m \times w$ matrix multiplication followed by truncation, our construction requires hw ring element of online communication per party in 1 round, whereas SPD$\mathbb{Z}_{2^k}$+ [15] requires $hw(2m+1)$ elements in 2 rounds. We improve online communicate cost by a factor of $2m+1$ and save one round of communication.

4.3 Secure Comparison

Many non-linear functions used in machine learning (*e.g.*, ReLU, MaxPool) involve comparison operations. The goal of secure comparison is to perform comparison operations on secret-shared data, ensuring that the output remains secret-shared as well.

We first consider comparing a secret-shared value $[x]$ with public value 0 to get $[z]$ where $z = (x < 0)$. The mask-and-compare technique, proposed by Damgård *et al.* [15] and Makri *et al.* [38], works as follows. In the preprocessing phase, the parties call $\mathcal{F}_{\mathsf{edaBits}}$ to get a shared random mask $[r]$ and its bit decomposition $\{[r_i]_2\}_{i=0}^{k-1} = ([r_1]_2, \ldots, [r_{k-1}]_2)$. In the online phase, the parties compute $[c] = [x] + [r]$ and open c in the clear, then use a bit-wise comparison circuit π_{BitLT} to compare the public value c and the secret-shared bits $\{[r_i]_2\}_{i=0}^{k-1}$, obtaining $[z]_2$ where $z = (c < r) = (x + r < r) = (x < 0)$. Finally, $[z]_2$ is converted to $[z]$ using a bit-to-arithmetic conversion functionality $\mathcal{F}_{\mathsf{B2A}}$. The procedure for standard bit-wise comparison π_{BitLT} is described in Appendix B.3, and the bit-to-arithmetic conversion functionality $\mathcal{F}_{\mathsf{B2A}}$ is described in Appendix A.4.

Now we develop our LTZ (less-than-zero) procedure π_{LTZ} in the context of circuit-dependent preprocessing. Parties have input $[\lambda_x], \Delta_x$ where $\Delta_x = \lambda_x + x$ and wish to obtain $[\lambda_z], \Delta_z$ where $z = (x < 0)$ and $\Delta_z = \lambda_z + z$.

To apply the mask-and-compare technique, the parties must open the value $[c] = [x] + [r] = \Delta_x - [\lambda_x] + [r]$. Let $\delta_x = r - \lambda_x$, observe that $[\delta_x] = [r] - [\lambda_x]$ can be computed and opened in the preprocessing phase. Consequently, $c = \Delta_x + (r - \lambda_x) = \Delta_x + \delta_x$ can be computed *locally* in the online phase. This approach effectively shifts the opening of the masked value to the preprocessing phase. Furthermore, if there are multiple LTZ gates in the circuit, then the opening of these values can be batched in a single round, saving communication rounds. For details, see the LTZ procedure π_{LTZ} described below.

On the use of the additional random mask $[r]$. Unlike the multiplication-then-truncation procedure in §4.1, where $[\lambda]$-values are utilized as truncation pairs, we do not reuse $[\lambda_x]$ for the mask-and-compare approach. The reason for this is that the bit-wise comparison procedure π_{BitLT} requires the knowledge of shared bit decomposition of the random mask. However, since the value $[\lambda_x]$ is determined prior to the preprocessing phase of π_{LTZ}, obtaining the shared bit decomposition of λ_x requires an expensive protocol [5]. Hence, we resort to generating an additional random mask $[r]$ using the relatively cheap $\mathcal{F}_{\mathsf{edaBits}}$.

Procedure 3: π_{LTZ}

For an LTZ gate with input wire x, to get the comparison result $z = (x < 0)$ on wire z:

Preprocessing phase:

1. Parties call $\mathcal{F}_{\mathsf{edaBits}}$ on input k to get $[r]$ and its bit decomposition $\{[r_i]_2\}_{i=0}^{k-1} = ([r_1]_2, \ldots, [r_{k-1}]_2)$.
2. Parties compute $[\delta_x] = [r] - [\lambda_x]$, then open δ_x.
3. Parties call $\mathcal{F}_{\mathsf{Prep}}$.Rand to get $[\lambda_z]$ where $\lambda_z \xleftarrow{\$} \mathbb{Z}_{2^k}$.

Online phase:

1. Parties call $\pi_{\mathsf{BitLT}}(\Delta_x + \delta_x, \{[r_i]_2\}_{i=0}^{k-1})$ to get $[z]_2$.
2. Parties call $\mathcal{F}_{\mathsf{B2A}}$ on input $[z]_2$ to get $[z]$.
3. Parties locally compute $[\Delta_z] = [\lambda_z] + [z]$.
4. Parties open Δ_z to get Δ_z in the clear.

Further optimization for the preprocessing phase. If procedure π_{LTZ} is to be executed right after procedure $\pi_{\mathsf{MultTrunc}}$, which is often the case in neural networks (*e.g.*, ReLU function after matrix multiplication), then the preprocessing phase can be further optimized. Note that the bit decomposition $([\lambda_{x,0}]_2, \ldots, [\lambda_{x,k-1}]_2)$ is already known in the preprocessing phase of $\pi_{\mathsf{MultTrunc}}$, hence it can serve as the random mask in the mask-and-compare approach, and $[r]$ is no longer needed. In the online phase, the paries replace the first step with $\pi_{\mathsf{BitLT}}(\Delta_x, \{[\lambda_{x,i}]_2\}_{i=0}^{k-1})$. Correctness holds since $(x < 0) = (\Delta_x - \lambda_x < 0) = (\Delta_x < \lambda_x)$. As a result, the call to $\mathcal{F}_{\mathsf{edaBits}}$ and the opening of δ_x is no longer needed.

Comparison of two secret values. To compare two secret values on wires x, y and obtain secret $(x < y)$ on wire z, the parties can simply invoke π_{LTZ} on input $(x - y)$.

Online communication cost. The online phase involves an opening of a $(s+k)$-bit ring element, an opening of a $(s+1)$-bit ring element (in $\mathcal{F}_{\mathsf{B2A}}$), and an invocation of π_{BitLT} on length-k inputs, which has $2k-2$ AND gates in $\log k$ rounds. Hence online communication cost for each party is $2(2k-2)(s+1) + (s+1) + (s+k) = 4ks + 5k - 2s - 3$ bits.

4.4 The Full Protocol

To formally state and prove the security of the procedures proposed above, we integrate them into a preprocessing protocol Π_{PrepPPML} and an online protocol $\Pi_{\mathsf{OnlinePPML}}$. They are put in Appendix C due to space constraints, and we briefly describe them below.

The preprocessing protocol Π_{PrepPPML} consists all of the components of $\Pi_{\mathsf{PrepArith}}$, in addition to the preprocessing phases of procedures $\pi_{\mathsf{MultTrunc}}$, $\pi_{\mathsf{DotProduct}}$, and π_{LTZ}. Similarly, the online protocol $\Pi_{\mathsf{OnlinePPML}}$, consists all of the components of $\Pi_{\mathsf{OnlineArith}}$, along with the online phases of procedures $\pi_{\mathsf{MultTrunc}}$, $\pi_{\mathsf{DotProduct}}$, and π_{LTZ}.

We define the corresponding functionalities $\mathcal{F}_{\mathsf{PrepPPML}}$ and $\mathcal{F}_{\mathsf{OnlinePPML}}$, they are described in Appendix A.3.

Now we state the security of our protocols. The proofs of the following theorems are given in the full version.

Theorem 4. *The protocol Π_{PrepPPML} securely implements the functionality $\mathcal{F}_{\mathsf{PrepPPML}}$ against a static, malicious adversary corrupting up to $n-1$ parties in the ($\mathcal{F}_{\mathsf{Prep}}$,$\mathcal{F}_{\mathsf{edaBits}}$)-hybrid model.*

Theorem 5. *The protocol $\Pi_{\mathsf{OnlinePPML}}$ securely implements the functionality $\mathcal{F}_{\mathsf{OnlinePPML}}$ against a static, malicious adversary corrupting up to $n-1$ parties in the ($\mathcal{F}_{\mathsf{PrepPPML}}$,$\mathcal{F}_{\mathsf{B2A}}$)-hybrid model.*

On the reactiveness of our protocols. Our constructions make use of circuit-dependent preprocessing, hence parties cannot modify circuit structure during the online phase. Nevertheless, our functionalities and protocols are reactive, in the sense that input gates and output gates are allowed in the midst of the circuit. Parties can learn partial outputs from the output gates before providing new inputs (which may depend on the outputs) to the input gate.

5 Applications to Machine Learning

In this section, we show how to apply privacy-preserving techniques to two types of popular ML models: SVMs and NNs, on top of our constructions in §3 and §4.

5.1 Support Vector Machines

SVMs are widely used for classification purposes. They can serve as binary classifiers or be extended to handle multiple categories. Concretely an SVM is a function $f : \mathbb{R}^n \to \mathbb{Z}_q$ defined as

$$f(\vec{x}) = \underset{i\in\{1,\ldots,q\}}{\operatorname{argmax}} \left(\mathbf{W}[i]\cdot\vec{x}+\vec{b}[i]\right)$$

where n is the dimension of the feature space and q is the number of categories. The input is $\vec{x} \in \mathbb{R}^n$. The matrix $\mathbf{W} \in \mathbb{R}^{q\times n}$ and the vector $\vec{b} \in \mathbb{R}^n$ are the parameters. Note that $\mathbf{W}[i]$ denotes the i-th row of $\mathbf{W}$.

To implement SVM inference, notice that it consists of only matrix multiplication and comparison operations, which can be efficiently implemented using the corresponding procedures $\pi_{\mathsf{DotProduct}}$ and π_{LTZ} in §4.

5.2 Neural Networks

We mainly focus on the inference of convolutional neural networks (CNN), which consists of fully connected layers, convolution layers, batch normalization layers, pooling layers, and activation functions. We assume familiarity with these concepts and refer the readers to a standard deep learning textbook (*e.g.* [42]) for details.

Linear layers. Fully connected layers, convolution layers, and batch normalization layers are linear layers, hence can be implemented using vector dot product.

Pooling layers. We consider two common kinds of pooling layers: max pooling and average pooling. Average pooling can be implemented using the addition and multiplication procedure, and max pooling needs comparison operations, which are accomplished by procedure π_{LTZ}.

Activation functions. We mainly consider the ReLU function $\mathrm{ReLU}(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$. For LeNet [34] we replace the legacy tanh function with ReLU since this enables faster training [32] and is easier to compute. To compute ReLU, notice that $\mathrm{ReLU}(x) = (x > 0)\cdot x = (1-(x<0))\cdot x$, parties can invoke π_{LTZ} on input x followed by multiplying $(1-(x<0))$ by x to obtain the result.

6 Evaluations

6.1 Experiment Setup

We implement our protocols in C++.[3] To compare with the state of the art SPD$\mathbb{Z}_{2^k}$+ [15], we implement the evaluations for SPD$\mathbb{Z}_{2^k}$+ in the MP-SPDZ framework [23]. Although both SPD$\mathbb{Z}_{2^k}$+ and MD-ML support multi-party computation, we mainly focus on 2PC scenario in the evaluations.

Execution environment. All of the experiments in this paper are performed on two cloud servers running Ubuntu 22.04 LTS. Both servers are equipped with 16 vCPUs and 32 GB of RAM. We consider two different network settings, LAN and WAN. The network parameters are simulated using the Linux `tc` command. The bandwidth between the parties are 10 Gbps (LAN) and 100 Mbps (WAN), respectively. The round-trip time (rtt) are 0.1ms (LAN) and 100ms (WAN), respectively.

Concrete parameters. Parameters $s = k = 64$ are used for the SPD$\mathbb{Z}_{2^k}$ secret-sharing scheme [9] in the evaluations of both MD-ML and SPD$\mathbb{Z}_{2^k}$+ [15]. For fixed point arithmetic in MD-ML, we use $d = 20$ for the number of fractional bits.

[3] Available at https://github.com/NemoYuan2008/MD-ML.

Datasets and ML models. The datasets used are MNIST [35], CIFAR-10 [31], Tiny ImageNet [33], and ImageNet [47]. The ML models used are SVM, LeNet [34], AlexNet [32], and ResNet-18 [22].

Metrics. We measure the total communication including all the messages sent by the paries (including TCP/IP headers, *etc.*). We measure the end-to-end execution time including the time of computation and network communication. The improvement factor of MD-ML over SPD$\mathbb{Z}_{2^k}$+ is included in the tables.

6.2 Microbenchmarks on the Building Blocks

6.2.1 Benchmarks of the online phase

We benchmark the online phase of the following building blocks for both MD-ML and SPD$\mathbb{Z}_{2^k}$+:

- Dot product of vector length 65536 (Table 4).
- Secure comparison (LTZ) of a batch of 65536 secret values (Table 4).
- Matrix multiplication of different sizes (Table 5).
- Tensor 2D convolution of different sizes (Table 6).

The execution time and communication costs of the online phase are measured and listed in the tables. Note that truncation operations are included in the benchmarks of multiplication, since they are needed in the application of PPML. The time and communication cost for the input phase is also included in the statistics.

The experimental results demonstrate the significant superiority of our efficient vector dot product and truncation protocols. Concretely, for linear operations (including convolution), we are 4.0–20.0× faster in LAN, 6.0–73.5× faster in WAN, and 1.7–8.8× more communication-efficient compared to SPD$\mathbb{Z}_{2^k}$+.

6.2.2 Benchmarks of the preprocessing phase

We benchmark the preprocessing phase of the following building blocks for both MD-ML and SPD$\mathbb{Z}_{2^k}$+ (Table 7):

- Dot product of vector length 65536.
- Multiplication with truncation of a batch of 1024 values.
- Secure comparison (LTZ) of a batch of 1024 values.

These correspond to the procedures $\pi_{\mathsf{DotProduct}}$, $\pi_{\mathsf{MultTrunc}}$, and π_{LTZ} from §4. Truncation and comparison operations are evaluated in a batch of 1024, aligning with the batch size of 1024 used in the edaBits generation implemented in the MP-SPDZ framework [23].

As shown by Table 7, the preprocessing cost of $\pi_{\mathsf{MultTrunc}}$ and π_{LTZ} are nearly the same as SPD$\mathbb{Z}_{2^k}$+, while the preprocessing cost of our vector dot product procedure is slightly higher than SPD$\mathbb{Z}_{2^k}$+. The reason is that parties need to generate the $[\lambda]$-values and open the δ-values in addition to the multiplication triples. This increase is very minimal compared to the significant efficiency enhancement of the online phase.

6.3 Benchmarks on ML models

We implement the following evaluations for online benchmarking:[4]

- SVM inference on the MNIST, CIFAR-10, and Tiny ImageNet datasets with both MD-ML and SPD$\mathbb{Z}_{2^k}$+ (Table 8).
- LeNet inference on the MNIST, CIFAR-10, and Tiny ImageNet datasets with both MD-ML and SPD$\mathbb{Z}_{2^k}$+ (Table 9).
- AlexNet inference on the CIFAR-10, Tiny ImageNet, and ImageNet datasets with both MD-ML and SPD$\mathbb{Z}_{2^k}$+ (Table 10).
- ResNet-18 inference on the CIFAR-10 dataset with MD-ML (Table 11).

The communication and the running time of the online phase are measured in both LAN and WAN settings.

Significant improvement of MD-ML over SPD$\mathbb{Z}_{2^k}$+ is observed from the tables. For LeNet inference, we are 3.4–9.2× faster in LAN, 9.7–31.7× faster in WAN, and 2.0–2.9× more communication-efficient. For AlexNet inference, we are 8.0–11.0× faster in LAN, 93.3–157.7× faster in WAN, and 9.8–23.8× more communication-efficient. For ResNet-18 inference on CIFAR-10, we are capable of completing the online phase of inference in 25.8s in LAN and 363.9s in WAN.

As the size of the neural network and dataset increases, we observe significant enhancements in online efficiency, owing to the utilization of our constant-communication dot product protocol and various efficient building blocks.

6.4 Accuracy

Table 12 compares the inference accuracy on MNIST dataset obtained by our MD-ML framework and by plaintext computation, using pre-trained SVM and LeNet models.

The primary sources of errors encountered in MD-ML can be attributed to the following factors: (1) the utilization of fixed-point arithmetic, (2) the introduction of errors through probabilistic truncation, and (3) the errors arising from comparison operations. The experimental results indicate that these errors are acceptable, with our relative error being 0.153%–0.435% compared to plaintext computation.

[4] In practice SVM and LeNet are not capable of classification of relatively large datasets such as Tiny ImageNet. Our objective in conducting the respective experiments is to showcase the performance of both frameworks.

Table 4: Online communication and running time of dot product and comparison

Operations†	LAN Time			WAN Time			Communication		
	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor
Dot product	2.5 ms	46.8 ms	18.7×	614 ms	4050 ms	6.6×	2.01 MB	2.10 MB	1.04×
LTZ	278 ms	1271 ms	4.57×	14.0 s	21.0 s	1.5×	132 MB	118.5 MB	0.9×

†Dot product is evaluated for vectors of length 65536. LTZ is evaluated for a batch of 65536 values.

Table 5: Online communication and running time of matrix multiplication

Shape† h,m,w	LAN Time			WAN Time			Communication		
	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor
1000, 2048, 1	35.3 ms	717 ms	20.0×	2.80 s	205.8 s	73.5×	31.4 MB	65.55 MB	2.1×
32, 32, 32	0.77 ms	9.05 ms	11.8×	0.429 s	7.84 s	18.3×	0.632 MB	1.065 MB	1.7×

† Input shape: a $h \times m$ matrix multiplied by a $m \times w$ matrix.

Table 6: Online communication and running time of 2D convolution

HW,C,M,h,s†	LAN Time			WAN Time			Communication		
	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor
28×28, 1, 1, 3, 1	1 ms	7.2 ms	7.2×	401 ms	2409 ms	6.0×	24.4 KB	210.6 KB	8.6×
32×32, 3, 1, 3, 1	2.15 ms	8.6 ms	4.0×	405 ms	3013 ms	7.4×	31.9 KB	280.3 KB	8.8×

† HW stands for the input shape, C the number of channels, M the number of kernels, $h \times h$ the kernel size, s the convolution stride.

Table 7: Preprocessing communication and running time of building blocks

Operation†	LAN time			WAN time			Communication		
	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor
MultTrunc	2.191 s	2.189 s	0.999×	436.991 s	436.383 s	0.999×	162.294 MB	162.261 MB	1.0000×
LTZ	2.388 s	2.390 s	1.001×	435.234 s	434.636 s	0.999×	165.096 MB	165.079 MB	0.9999×
Dot prod.	8.065 s	6.246 s	0.775×	283.548 s	230.505 s	0.813×	1270.23 MB	1124.39 MB	0.8852×

†Dot product is evaluated for vectors of length 65536. MultTrunc and LTZ are evaluated for a batch of 1024 values.

Table 8: Online communication and running time of SVM inference on different datasets

Dataset	LAN Time			WAN Time			Communication		
	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor
MNIST	0.92 ms	4 ms	4.3×	641 ms	2208 ms	3.4×	137 KB	257.06 KB	1.9×
CIFAR-10	2.2 ms	21 ms	9.5×	681 ms	3670 ms	5.4×	518 KB	1006.79 KB	1.9×
Tiny ImageNet	47 ms	600 ms	12.8×	3321 ms	45041 ms	13.6×	37.8 MB	78.65 MB	2.1×

Table 9: Online communication and running time of LeNet inference on different datasets

Dataset	LAN Time			WAN Time			Communication		
	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor
MNIST	80 ms	274 ms	3.4×	9.82 s	95.647 s	9.7×	9.46 MB	18.99 MB	2.0×
CIFAR-10	89 ms	399 ms	4.5×	10.14 s	117.309 s	11.5×	13.1 MB	34.41 MB	2.6×
Tiny ImageNet	196 ms	1798 ms	9.2×	13.968 s	443.397 s	31.7×	65.5 MB	187.62 MB	2.9×

Table 10: Online communication and running time of AlexNet inference on different datasets

Dataset	LAN Time			WAN Time			Communication		
	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor	Ours	SPD$\mathbb{Z}_{2^k}$+	Factor
CIFAR-10	0.82 s	6.80 s	8.3×	34.88 s	3254.7 s	93.3×	241.51 MB	2364.82 MB	9.8×
Tiny ImageNet	2.06 s	16.40 s	8.0×	53.89 s	6774.6 s	125.7×	405.00 MB	8274.95 MB	20.4×
ImageNet	7.38 s	81.35 s	11.0×	188.92 s	29785.2 s	157.7×	1319.31 MB	31447.70 MB	23.8×

Table 11: Online Performance of ResNet-18 with MD-ML

Model and Dataset	LAN	WAN	Comm.
ResNet-18 on CIFAR-10	25.8 s	362.9 s	4.15 GB

Table 12: Inference Accuracy on MNIST dataset

Model	Plaintext	MD-ML	Relative Error
SVM	94.42%	94.28%	0.153%
LeNet	98.86%	98.43%	0.435%

7 Conclusion

In this work, we introduce MD-ML, a framework for maliciously secure dishonest majority PPML. We have developed a series of efficient building blocks such as dot product, multiplication with truncation, and comparison. We have significantly improved the performance in terms of online communication cost compared to SPD$\mathbb{Z}_{2^k}$+ [15]. Finally, we demonstrate the practicality and effectiveness of our constructions by implementing SVM, LeNet, AlexNet, and ResNet with MD-ML. We believe that MD-ML will promote the real-world applications of maliciously secure dishonest majority MPC and PPML.

There are several potential future directions for our research: (1) exploring the benefits of flexible corruption (§3.3.2), and (2) improving the efficiency of our constructions using orthogonal optimizations such as hardware acceleration.

Acknowledgment

We would like to thank the anonymous reviewers of USENIX Security 2024 for their valuable comments. This work was supported in part by the National Key Research and Development Project 2020YFA0712300, and National Natural Science Foundation of China (Grant No. 62272294).

References

[1] Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, and Chen Yuan. Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings. In Dennis Hofheinz and Alon Rosen, editors, *Theory of Cryptography*, pages 471–501, Cham, 2019. Springer International Publishing.

[2] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 420–432, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[3] Aner Ben-Efraim, Michael Nielsen, and Eran Omri. Turbospeedz: Double your online SPDZ! Improving SPDZ using function dependent preprocessing. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *Applied Cryptography and Network Security*, pages 530–549, Cham, 2019. Springer International Publishing.

[4] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptol-

ogy ePrint Archive, Paper 2000/067, 2000. https://eprint.iacr.org/2000/067.

[5] Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. In Juan A. Garay and Roberto De Prisco, editors, *Security and Cryptography for Networks*, pages 182–199, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[6] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. Astra: High throughput 3pc over rings with application to secure prediction. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, CCSW'19, page 81–92, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4pc framework for privacy preserving machine learning. In *Proceedings 2020 Network and Distributed System Security Symposium*. Internet Society, 2020.

[8] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority mpc for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 34–64, Cham, 2018. Springer International Publishing.

[9] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. Spd$\mathbb{Z}_{2^k}$: Efficient mpc mod 2^k for dishonest majority. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 769–798, Cham, 2018. Springer International Publishing.

[10] Ronald Cramer, Matthieu Rambaud, and Chaoping Xing. Asymptotically-good arithmetic secret sharing over $\mathbb{Z}/p^\ell\mathbb{Z}$ with strong multiplication and its applications to efficient mpc. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021*, pages 656–686, Cham, 2021. Springer International Publishing.

[11] Anders Dalskov, Daniel Escudero, and Marcel Keller. Secure evaluation of quantized neural networks. *Proceedings on Privacy Enhancing Technologies Symposium*, 2020(4):355–375, 2020.

[12] Anders Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-Majority Four-Party secure computation with malicious security. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2183–2200. USENIX Association, August 2021.

[13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure mpc for dishonest majority – or: Breaking the spdz limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security – ESORICS 2013*, pages 1–18, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[14] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pages 643–662, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[15] Ivan Damgård, Daniel Escudero, Tore Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure mpc over rings with applications to private machine learning. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1102–1120, 2019.

[16] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY — A framework for efficient mixed-protocol secure two-party computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8–11, 2015*. The Internet Society, 2015.

[17] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for mpc over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020*, pages 823–852, Cham, 2020. Springer International Publishing.

[18] Daniel Escudero, Vipul Goyal, Antigoni Polychroniadou, Yifan Song, and Chenkai Weng. Superpack: Dishonest majority mpc with constant online communication. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages 220–250, Cham, 2023. Springer Nature Switzerland.

[19] Daniel Escudero, Chaoping Xing, and Chen Yuan. More efficient dishonest majority secure computation over $\mathbb{Z}_{2^k}$ via galois rings. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, pages 383–412, Cham, 2022. Springer Nature Switzerland.

[20] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game, or a completeness theorem for protocols with honest majority. In *STOC'87*, pages 218–229, 1987.

[21] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Sharing transformation and dishonest majority mpc with packed secret sharing. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, pages 3–32, Cham, 2022. Springer Nature Switzerland.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[23] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1575–1590, New York, NY, USA, 2020. Association for Computing Machinery.

[24] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 830–842, New York, NY, USA, 2016. Association for Computing Machinery.

[25] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making spdz great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 158–189, Cham, 2018. Springer International Publishing.

[26] Marcel Keller and Ke Sun. Secure quantized training for deep learning. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 10912–10938. PMLR, 17–23 Jul 2022.

[27] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. Crypten: Secure multi-party computation meets machine learning. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 4961–4973. Curran Associates, Inc., 2021.

[28] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: Super-fast and robust Privacy-Preserving machine learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2651–2668. USENIX Association, August 2021.

[29] Nishat Koti, Shravani Patil, Arpita Patra, and Ajith Suresh. MPClan: Protocol suite for privacy-conscious computations. *Journal of Cryptology*, 36(3):22, 2023.

[30] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. Tetrad: Actively secure 4pc for secure training and inference. In *Proceedings 2022 Network and Distributed System Security Symposium*. Internet Society, 2022.

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

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, may 2017.

[33] Ya Le and Xuan Yang. Tiny imagenet visual recognition challenge. *CS 231N*, 7(7):3, 2015.

[34] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[35] Yann LeCun, Corinna Cortes, and Christopher J Burges. Mnist handwritten digit database. 2010. *URL http://yann. lecun. com/exdb/mnist*, 7(23):6, 2010.

[36] Yehuda Lindell. How to simulate it - a tutorial on the simulation proof technique. Cryptology ePrint Archive, Paper 2016/046, 2016. https://eprint.iacr.org/2016/046.

[37] Fengrun Liu, Xiang Xie, and Yu Yu. Scalable multi-party computation protocols for machine learning in the honest-majority setting. To be presented at USENIX'24.

[38] Eleftheria Makri, Dragos Rotaru, Frederik Vercauteren, and Sameer Wagh. Rabbit: Efficient comparison for secure multi-party computation. In Nikita Borisov and Claudia Diaz, editors, *Financial Cryptography and Data Security*, pages 249–270, Berlin, Heidelberg, 2021. Springer Berlin Heidelberg.

[39] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 35–52, New York, NY, USA, 2018. Association for Computing Machinery.

[40] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38, 2017.

[41] L. L. Ng and S. M. Chow. Sok: Cryptographic neural-network computation. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 497–514, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[42] Michael Nielsen. *Neural Networks and Deep Learning*. Determination press, 2015.

[43] Emmanuela Orsini. Efficient, actively secure mpc with a dishonest majority: A survey. In Jean Claude Bajard and Alev Topuzoğlu, editors, *Arithmetic of Finite*

Fields, pages 42–71, Cham, 2021. Springer International Publishing.

[44] Emmanuela Orsini, Nigel P. Smart, and Frederik Vercauteren. Overdrive2k: Efficient secure mpc over $\mathbb{Z}_{2^k}$ from somewhat homomorphic encryption. In Stanislaw Jarecki, editor, *Topics in Cryptology – CT-RSA 2020*, pages 254–283, Cham, 2020. Springer International Publishing.

[45] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: Improved Mixed-Protocol secure Two-Party computation. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2165–2182. USENIX Association, August 2021.

[46] Arpita Patra and Ajith Suresh. BLAZE: Blazing fast privacy-preserving machine learning. In *Proceedings 2020 Network and Distributed System Security Symposium*. Internet Society, 2020.

[47] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[48] SecureSCM. Security analysis. Deliverable D9.2, EU FP7 Project Secure Supply Chain Management (SecureSCM), 2009.

[49] Sijun Tan, Brian Knott, Yuan Tian, and David J. Wu. Cryptgpu: Fast privacy-preserving machine learning on the gpu. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1021–1038, 2021.

[50] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-Party Secure Computation for Neural Network Training. *Proceedings on Privacy Enhancing Technologies*, 2019.

[51] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-Majority Maliciously Secure Framework for Private Deep Learning. *Proceedings on Privacy Enhancing Technologies*, 2021.

[52] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. Piranha: A GPU platform for secure computation. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 827–844, Boston, MA, August 2022. USENIX Association.

[53] Andrew C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 160–164, 1982.

A Functionalities

A.1 SPD$\mathbb{Z}_{2^k}$ Preprocessing Functionality

Functionality $\mathcal{F}_{\mathsf{Prep}}$ is taken from [9] and is described below.

Functionality 1: $\mathcal{F}_{\mathsf{Prep}}$

Input: On input (Input, P_i) from all parties, sample random $r \xleftarrow{\$} \mathbb{Z}_{2^k}$ and generate $[r]$. If P_i is corrupted, instead let the adversary choose the sharing $[r]$. Then send r to P_i, and distributes the sharing $[r]$ to all parties.
Triple: On input (Triple) from all parties, sample two random values $a, b \xleftarrow{\$} \mathbb{Z}_{2^k}$, compute $c = ab \bmod 2^k$, generate $[a], [b], [c]$, and distributes the sharings $[a], [b], [c]$ to all parties.
Rand: On input (Rand) from all parties, sample random value $r \xleftarrow{\$} \mathbb{Z}_{2^k}$, generate $[r]$, and distributes the sharing $[r]$ to all parties.

A.2 Our Functionalities for Arithmetic Circuits

We provide formal definitions of our functionalities for arithmetic circuits (§3) in this section.

Functionality 2: $\mathcal{F}_{\mathsf{PrepArith}}$

Input: On input $(\mathsf{Input}, P_i, id)$ from the parties, where id is a fresh, valid identifier for the input wire of P_i, the functionality samples $\lambda_x \xleftarrow{\$} \mathbb{Z}_{2^k}$ and generates $[\lambda_x]$. If P_i is corrupted, instead let the adversary choose $[\lambda_x]$. Then, the functionality stores $(id, [\lambda_x])$. Finally, the functionality sends λ_x to P_i and distributes $[\lambda_x]$ to all parties.
Add: On input $(\mathsf{Add}, id_1, id_2, id_3)$ from the parties, the functionality retrieves (if present in memory) the values $(id_1, [\lambda_x]), (id_2, [\lambda_y])$ and stores $(id_3, [\lambda_z])$ where $\lambda_z = \lambda_x + \lambda_y$.
Multiply: On input $(\mathsf{Multiply}, id_1, id_2, id_3)$ from the parties, the functionality retrieves (if present in memory) the values $(id_1, [\lambda_x]), (id_2, [\lambda_y])$. The functionality then samples $a, b, \lambda_z \xleftarrow{\$} \mathbb{Z}_{2^k}$, computes $c = ab$, generates $[a], [b], [c], [\lambda_z]$, and distributes them to the parties. Finally, the functionality computes $\delta_x = a - \lambda_x, \delta_y = b - \lambda_y$ and stores $(id_3, [\lambda_z])$ and δ_x, δ_y.
Output: The functionality sends all the stored δ-values to the adversary, then waits for a message Abort or $\mathsf{Proceed}$ from the adversary: if it sends Abort then the functionality aborts, otherwise the functionality sends the δ-values to the parties.

Functionality 3: $\mathcal{F}_{\mathsf{OnlineArith}}$

Initialize: Receive (init, k) from all parties.
Input: On input $(\mathsf{Input}, P_i, id, x)$ from P_i and input (Input, P_i) from other parties, where id is a fresh identifier, store (id, x).
Add: On input $(\mathsf{Add}, id_1, id_2, id_3)$ from all the parties (where id_1 and id_2 are present in memory), retrieve (id_1, x), (id_2, y) and store $(id_3, x+y)$.
Multiply: On input $(\mathsf{Mult}, id_1, id_2, id_3)$ from all the parties (where id_1, id_2 are present in memory), retrieve $(id_1, x), (id_2, y)$ and store (id_3, xy).
Output: On input (Output, id) from all the parties (where id is present in memory), retrieve (id, y) and output it to the adversary. Wait for an input Proceed or Abort from the adversary. If this is Proceed then send y to all parties, otherwise abort.

A.3 Our Functionalities for PPML

We provide formal definitions of our functionalities for PPML (§4) in this section.

Functionality 4: $\mathcal{F}_{\mathsf{PrepPPML}}$

Functionality $\mathcal{F}_{\mathsf{PrepPPML}}$ has the same commands as $\mathcal{F}_{\mathsf{PrepArith}}$, in addition to the following:
MultTrunc: On input $(\mathsf{MultTrunc}, id_1, id_2, id_3)$ from all parties, retrieve (if present in memory) the values $(id_1, [\lambda_x]), (id_2, [\lambda_y])$. Then, sample $a, b, \lambda_z \xleftarrow{\$} \mathbb{Z}_{2^k}$, compute $c = ab$, generate $[a], [b], [c], [\lambda_z]$, generate the shared bit decomposition $([\lambda_{z,0}]_2, \ldots, [\lambda_{z,k-1}]_2)$, generate $\{[\lambda_{z',i}]\}_{i=d}^{k-1}$, and distribute all of the above sharings to the parties. Finally, compute $\delta_x = a - \lambda_x, \delta_y = b - \lambda_y$, store $(id_3, [\lambda_z])$ and δ_x, δ_y.
DotProduct: On input $(\mathsf{DotProduct}, id_1, id_2, id_3)$ from all parties (where id_1, id_2 are present in memory which represent vectors of equal length m), retrieve (if present in memory) the values $(id_1, [\overrightarrow{\lambda_x}]), (id_2, [\overrightarrow{\lambda_y}])$. Then, sample $\vec{a}, \vec{b}, \xleftarrow{\$} \mathbb{Z}_{2^k}^m$ and $\lambda_z \xleftarrow{\$} \mathbb{Z}_{2^k}$, compute $\vec{c}$ such that $\vec{c}[i] = \vec{a}[i]\vec{b}[i]$. Then, generate $[\vec{a}], [\vec{b}], [\vec{c}], [\lambda_z]$, and distributes them to the parties. Finally, compute $\overrightarrow{\delta_x} = \vec{a} - \overrightarrow{\lambda_x}, \overrightarrow{\delta_y} = \vec{b} - \overrightarrow{\lambda_y}$, store $(id_3, [\lambda_z])$ and $\overrightarrow{\delta_x}, \overrightarrow{\delta_y}$.
LTZ: On input $(\mathsf{LTZ}, id_1, id_2)$ from all parties (where id_1 is present in memory), retrieve (if present in memory) the value $(id_1, [\lambda_x])$. Then, sample $r \xleftarrow{\$} \mathbb{Z}_{2^k}$, generate and distribute $[r]$ to the parties. Then, compute $\delta_x = r - \lambda_x$ and store δ_x. Finally, sample $\lambda_z \xleftarrow{\$} \mathbb{Z}_{2^k}$, generate and distribute $[\lambda_z]$ to the parties, and store (id_2, λ_z).

Functionality 5: $\mathcal{F}_{\mathsf{OnlinePPML}}$

Functionality $\mathcal{F}_{\mathsf{OnlinePPML}}$ has the same commands as $\mathcal{F}_{\mathsf{OnlineArith}}$, in addition to the following:
MultTrunc: On input $(\mathsf{MultTrunc}, id_1, id_2, id_3)$ from all parties (where id_1, id_2 are present in memory), retrieve $(id_1, x), (id_2, y)$ and store $(id_3, xy/2^d)$.
DotProduct: On input $(\mathsf{DotProduct}, id_1, id_2, id_3)$ from all parties (where id_1, id_2 are present in memory which represent vectors of equal length), retrieve $(id_1, \vec{x}), (id_2, \vec{y})$ and store $(id_3, \vec{x} \cdot \vec{y})$.
LTZ: On input $(\mathsf{LTZ}, id_1, id_2)$ from all parties (where id_1 is present in memory), retrieve (id_1, x) and store $(id_2, (x < 0))$.

A.4 Functionalities for edaBits and B2A

Functionality $\mathcal{F}_{\mathsf{edaBits}}$ is taken from [17].

Functionality 6: $\mathcal{F}_{\mathsf{edaBits}}$

On input $(\mathsf{edaBits}, \ell)$ from all parties, sample uniformly random values $(r_0, \ldots, r_{\ell-1}) \in \mathbb{Z}_2^\ell$, compute $r = \sum_{i=0}^{\ell-1} 2^i r_i$, generate $\{[r_i]_2\}_{i=0}^{\ell-1}$ and $[r]$, then distributes the sharings to all parties.

Functionality $\mathcal{F}_{\mathsf{B2A}}$ is a standard conversion functionality.

Functionality 7: $\mathcal{F}_{\mathsf{B2A}}$

On input $(\mathsf{B2A}, [x]_2)$ from all parties, compute $x \in \{0,1\}$ according to $[x]_2$, then generate $[x]$ and distribute the sharing $[x]$ to all parties.

B Procedures

B.1 MAC Checking

The procedure π_{MACCheck} is taken from [9]. It relies on a coin tossing functionality $\mathcal{F}_{\mathsf{Rand}}$ to sample public random values for the parties.

Procedure 4: π_{MACCheck}

Procedure π_{MACCheck} takes as input a set of opened values $(x_1, x_2, \ldots, x_t)$ and checks the correctness of the corresponding MACs. Let x_i^j, m_i^j, α_i^j be P_j's share, MAC share and MAC key share for x_i.

1. Parties call $\mathcal{F}_{\mathsf{Rand}}$ to sample public random values $(r_1, r_2, \ldots, r_t) \xleftarrow{\$} \mathbb{Z}_{2^s}^t$ and compute $y = \sum_{i=1}^t r_i \cdot x_i \bmod 2^{k+s}$.
2. Each Party P_j computes $m^j = \sum_{i=1}^t r_i m_i^j$ and $z^j = m^j - \alpha^j \cdot y$, then it commits to z^j.

3. Parties open their commitments and verify that $\sum_{i=j}^{n} z^j = 0 \pmod{2^{k+s}}$. If the check fails then parties abort.

The failure probability of π_{MACCheck} is negligible in s, as stated in the following theorem.

Theorem 6 ([9, Theorem 1]). *Assume α is uniformly random to the adversary, if procedure π_{MACCheck} passes, then the values accepted by the parties are correct, except with probability at most $2^{-s+\log(s+1)}$.*

B.2 Dot Product with Truncation

We describe our dot-product-then-truncate procedure below.

Procedure 5: $\pi_{\mathsf{DotProductTrunc}}$

For a dot-product-then-truncate gate with input vectors of length m on wires $\vec{x}, \vec{y}$, to compute dot product and truncate the result by d bits on output wire z:

Preprocessing phase:

1. Parties call $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Triple}$ m times to get m triples, obtaining $[\vec{a}], [\vec{b}], [\vec{c}]$, where $([\vec{a}[i]], [\vec{b}[i]], [\vec{c}[i]])$ is an individual multiplication triple obtained in the i-th call of $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Triple}$.
2. Parties compute $[\overrightarrow{\delta_x}] = [\vec{a}] - [\overrightarrow{\lambda_x}]$, $[\overrightarrow{\delta_y}] = [\vec{b}] - [\lambda_y]$ and open $\overrightarrow{\delta_x}, \overrightarrow{\delta_y}$.
3. Parties call $\mathcal{F}_{\mathsf{edaBits}}$ on input $(k-d)$ to get $[\lambda_z]$ and its bit decomposition $\{[\lambda_{z,i}]_2\}_{i=0}^{k-d-1}$.
4. Parties call $\mathcal{F}_{\mathsf{edaBits}}$ on input d to get $[u]$ and its bit decomposition $\{[u_i]_2\}_{i=0}^{d-1}$.
5. Parties locally compute $[\lambda_{z'}] = 2^d \cdot [\lambda_z] + [u]$.

Online phase:

1. Parties locally compute $[\Delta_{z'}] = \sum_{i=1}^{m} \Big((\overrightarrow{\Delta_x}[i] + \overrightarrow{\delta_x}[i])(\overrightarrow{\Delta_y}[i] + \overrightarrow{\delta_y}[i]) - (\overrightarrow{\Delta_y}[i] + \overrightarrow{\delta_y}[i])[\vec{a}[i]] - (\overrightarrow{\Delta_x}[i] + \overrightarrow{\delta_x}[i])[\vec{b}[i]] + [\vec{c}[i]] \Big) + [\lambda_{z'}]$.
2. Parties open $\Delta_{z'}$ to get $\Delta_{z'}$ in the clear.
3. Parties locally compute $\Delta_z = \Delta_{z'} / 2^d$.

B.3 Bit-wise Comparison

The bit-wise comparison procedure π_{BitLT} is taken from [15]. The procedure uses a circuit $\pi_{\mathsf{Carry}}(x, \{[y_i]_2\}_{i=0}^{k-1}, u)$ to compute the carry bit of an addition between two integers $x, y \in \mathbb{Z}_{2^k}$, when the initial carry-in bit is $u \in \{0,1\}$. A circuit with $2k-2$ AND gates can be constructed using standard methods, and we refer to the CarryOutCin circuit described in [48].

The π_{BitLT} procedure is described below, we denote it as $\pi_{\mathsf{BitLT}}\left(x, \{[y_i]_2\}_{i=0}^{k}\right)$.

Procedure 6: π_{BitLT}

Input: A public value x, and the shared bit decomposition $\{[y_i]_2\}_{i=0}^{k-1} = ([y_0]_2, \ldots, [y_{k-1}]_2)$ of value y.
Output: A shared value $[z]_2$ where $z = (x < y)$.
The Procedure:

1. Parties set $[y'_i]_2 = 1 - [y_i]_2$ for $i = 0, 1, \ldots, k-1$.
2. Parties set $[z]_2 = 1 - \pi_{\mathsf{Carry}}(x, \{[y'_i]_2\}_{i=0}^{k-1}, 1)$.

C Protocols

We describe our full preprocessing protocol Π_{PrepPPML} and online protocol $\Pi_{\mathsf{OnlinePPML}}$ for PPML (§4) below.

Protocol 3: Π_{PrepPPML}

The parties proceed in topological order.
Input: Same as in $\Pi_{\mathsf{PrepArith}}$.
Add: Same as in $\Pi_{\mathsf{PrepArith}}$.
Multiply: Same as in $\Pi_{\mathsf{PrepArith}}$.
MultTrunc: Parties execute the preprocessing phase of $\pi_{\mathsf{MultTrunc}}$, the opening of the δ-values can be deferred and batched in the Output phase.
DotProduct: Parties execute the preprocessing phase of $\pi_{\mathsf{DotProduct}}$, the opening of the δ-values can be deferred and batched in the Output phase.
LTZ: Parties execute the preprocessing phase of π_{LTZ}, the opening of the δ-values can be deferred and batched in the Output phase.
Output: Same as in $\Pi_{\mathsf{PrepArith}}$.

Protocol 4: $\Pi_{\mathsf{OnlinePPML}}$

Initialize: Parties call $\mathcal{F}_{\mathsf{PrepPPML}}$ with the circuit to get the δ-values, the shared $[\lambda]$-values, and the multiplication triples for each gate.
Then, parties proceed in topological order.
Input: Same as in $\Pi_{\mathsf{OnlineArith}}$.
Add: Same as in $\Pi_{\mathsf{OnlineArith}}$.
Multiply: Same as in $\Pi_{\mathsf{OnlineArith}}$.
MultTrunc: Parties execute the online phase of $\pi_{\mathsf{MultTrunc}}$.
DotProduct: Parties execute the online phase of $\pi_{\mathsf{DotProduct}}$.
LTZ: Parties execute the online phase of π_{LTZ}.
Output: Same as in $\Pi_{\mathsf{OnlineArith}}$.