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

Boshi Yuan, Shixuan Yang, and Yongxiang Zhang, *Shanghai Jiao Tong University, China*; Ning Ding, Dawu Gu, and Shi-Feng Sun, *Shanghai Jiao Tong University, China*; *Shanghai Jiao Tong University (Wuxi) Blockchain Advanced Research Center*

https://www.usenix.org/conference/usenixsecurity24/presentation/yuan

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.
August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.
MD-ML: Super Fast Privacy-Preserving Machine Learning for Malicious Security with a Dishonest Majority

Boshi Yuan¹, Shixuan Yang¹, Yongxiang Zhang¹, Ning Ding¹,²,*, Dawu Gu¹,²,*, and Shi-Feng Sun¹,²

¹Shanghai Jiao Tong University, China
²Shanghai Jiao Tong University (Wuxi) Blockchain Advanced Research Center

{nymoyuan2008,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.

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-
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 breach the privacy of the data and the correctness of the results. In such cases, considering semi-honest or maliciously secure honest majority protocols 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, maliciously secure, dishonest majority ppML. 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 “SPDZ$_{2k}^+$” is adopted to denote the state of the art by Damgård et al. [15], since their protocols are based on the SPDZ$_{2k}$ protocol [9].

**Efficient Protocols for PPML.** We extend the circuit-dependent preprocessing technique of TurboSpeedz [3] to the ring $Z_{2k}$, 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):

- Multiplication with truncation (§4.4): 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 SPDZ$_{2k}$+ [15] which requires $2hw$ elements online communication for each party.

- Secure Comparison (§4.3): To enable computing nonlinear 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.

### Table 1: Comparison of theoretical online communication cost with SPDZ$_{2k}^+$ [15]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Ours</th>
<th>SPDZ$_{2k}^+$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplication</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Vector dot product†</td>
<td>1</td>
<td>2$m$</td>
</tr>
<tr>
<td>Matrix multiplication‡</td>
<td>hw</td>
<td>2$mw$</td>
</tr>
<tr>
<td>Matrix multiplication with truncation‡</td>
<td>hw</td>
<td>(2$m + 1$)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Online comm. of</th>
<th>Ours</th>
<th>SPDZ$_{2k}^+$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplication</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Vector dot product†</td>
<td>1</td>
<td>2$m$</td>
</tr>
<tr>
<td>Matrix multiplication‡</td>
<td>hw</td>
<td>2$mw$</td>
</tr>
<tr>
<td>Matrix multiplication with truncation‡</td>
<td>hw</td>
<td>(2$m + 1$)</td>
</tr>
</tbody>
</table>
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 SPDZ\[2]+ [15]. The highlights are summarized in Table 2.

![Table 2: Improvement over SPDZ\[2]+ [15] in terms of online execution time and communication for ML model inference](image)

<table>
<thead>
<tr>
<th>Model</th>
<th>Time</th>
<th>Comm.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LAN</td>
<td>WAN</td>
</tr>
<tr>
<td>SVM</td>
<td>4.3–12.8×</td>
<td>3.4–13.6×</td>
</tr>
<tr>
<td>LeNet</td>
<td>3.4–9.2×</td>
<td>9.7–31.7×</td>
</tr>
<tr>
<td>AlexNet</td>
<td>8.0–11.0×</td>
<td>93.3–157.7×</td>
</tr>
<tr>
<td>Resnet-18\†</td>
<td>25.8 s</td>
<td>362.9 s</td>
</tr>
</tbody>
</table>

\† ResNet-18 inference are not carried out for SPDZ\[2]+, 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 SPDZ\[2]-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 $\Delta$-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 preprocess- protocol $\Pi_{\text{PrepPPML}}$ and an online protocol $\Pi_{\text{OnlinePPML}}$. By defining the corresponding functionalities as $\mathcal{F}_{\text{PrepPPML}}$ and $\mathcal{F}_{\text{OnlinePPML}}$, 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 $F$. 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 SPDZ$_{2k}$ 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 $Z_{2k}$ by introducing a special packing technique for BGV encryption. Escudero et al. [19] achieve a lower amortized communication complexity for MPC over $Z_{2k}$ 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 SPDZ$_{2k}$ secret-sharing scheme. In §3 we give our protocols for arithmetic circuits over the ring $Z_{2k}$. 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 $Z_L$ the set of integers $\{0, 1, \ldots, L-1\}$, and use $\lambda \leftarrow Z_L$ to denote that $\lambda$ is uniformly random in $Z_L$. For a vector $\vec{x}$, denote by $\vec{x}[i]$ the $i$-th element of $\vec{x}$. The dot product of vectors $\vec{x}$ and $\vec{y}$ is denoted by $\vec{x} \cdot \vec{y}$. For a $k$-bit integer $\lambda$, we explicitly use $(\lambda_0, \ldots, \lambda_{k-1})$ to denote the bit decomposition of $\lambda$, where $\lambda_0$ is the least significant bit. This notation is employed to differentiate the individual bits of $\lambda$ 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.

| $P_1, P_2, \ldots, P_n$ | Parties performing secure computation |
| $Z_L$ | The set of integers $\{0, 1, \ldots, L-1\}$ |
| $\lambda \leftarrow Z_L$ | $\lambda$ is uniformly random in $Z_L$ |
| $\vec{x} \cdot \vec{y}$ | 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 $\lambda$ |
| $(x < y)$ | Comparison result of $x < y$ |
| $[x]$ | SPDZ$_{2k}$ arithmetic share of $x$ (§2.3) |
| $[x]_{2^k}$ | SPDZ$_{2k}$ binary share of $x \in Z_2$ (§2.3) |
| $k, s$ | Concrete parameters for SPDZ$_{2k}$ (§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 $F$ with some subscript), and the real world, instantiated by a protocol (denoted by $\Pi$ with some subscript). Protocols can invoke procedures (denoted by $\pi$ with some subscript), which are like protocols but not intended to instantiate a functionality. Instead they are used as subroutines inside other protocols that instantiate some functionality.

#### 2.3 The SPDZ$_{2k}$ Secret-sharing Scheme

We briefly recall the SPDZ$_{2k}$ secret-sharing scheme [9] in this section, which enables $n$ parties to compute over the ring $Z_{2^k}$ 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 $Z_{2^k+s}$, where $s = s - \log s$ is the statistical security parameter, but security and correctness is only guaranteed modulo $2^k$.

**Definition of SPDZ$_{2k}$ secret-sharing scheme** $[\cdot]_{2^k}.$ Assume each party $P_i$ (for $i = 1, 2, \ldots, n$) holds a uniformly random additive share $\alpha^i \leftarrow Z_{2^k}$ of a secret global MAC key $\alpha = \sum_{i=1}^n \alpha^i \mod 2^{k+s}$. An element $x \in Z_{2^k}$ is $[\cdot]_{2^k}$-shared if each party holds $x^i \in Z_{2^k+s}$ and $m^i \in Z_{2^k+s}$, where $x = \sum_{i=1}^n x^i \mod 2^k$ is the element being shared, and $m^i = \sum_{j=i+1}^n m^j \mod 2^{k+s}$ is the MAC, such that $m^i = (\sum_{j=i+1}^n x^j) \cdot \alpha \mod 2^{k+s}$. We denote $[x]_{2^k} = ((x^1, \ldots, x^n), (m^1, \ldots, m^n))$. 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 \([x] = ((x^1, \ldots, x^n), (m_1^1, \ldots, m_1^n), \ldots, (m_k^1, \ldots, m_k^n))\) and public constants \(a, b, c\), the linear combination \([z] = [ax] + b[\cdot] + c\) can be computed as \([z] = ([z^1, \ldots, z^n], (m_1^1, \ldots, m_1^n))\) where \(P_i\) locally computes \(z^i = ax^i + by^i + c, \quad \text{if } i = 1\) and

\[ m_i^z = am_i^x + bm_i^y + cx \text{ for } i = 1, \ldots, n. \]

**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 \([\bar{x}] = [x] + 2^s[r]\). Denote \(P_i\)'s share and MAC share on \([x]\) by \(\bar{x}^i\) and \(m_{1,i}\). Each party \(P_i\) broadcasts its own \(\bar{x}^i\) (but not \(m_{1,i}\)) and, upon receiving the values from other parties, computes \(x = \sum_{i=1}^n \bar{x}^i \mod 2^k\). To ensure correctness, the MAC \(m_{1,i}\) should be checked, but can be deferred and batched with other opened values. Procedure \(\text{MACCheck}\) 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, \(\text{MACCheck}\) 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 SPDZ\(_{2^k}\)-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 \(\mathbb{Z}_{2^k}\).

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 \((\lambda_z)\). If \(z\) is not the output of an addition gate, then \(\lambda_z\) is a uniformly random value. If \(z\) is the output of an addition gate with input wires \(x, y\), then \(\lambda_z\) 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 \(\langle \langle a \rangle, \langle b \rangle, \langle c \rangle \rangle\) and values \(\delta_x, \delta_y\), 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 \(\lambda_x = x + \lambda_x\) 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 \(\Delta = \lambda_x + \lambda_y\) locally. For a multiplication gate with input \((x, y)\), to get the result \(\langle z \rangle\), the parties can compute the output as \(\langle z \rangle = (\Delta + \delta_x)(\Delta + \delta_y) - (\Delta + \delta_y)(\langle a \rangle - (\Delta_x + \delta_x))\), where \(\Delta_x + \delta_x = y + b, \text{ hence for the right-hand-side of } \langle z \rangle\) we have: \((\lambda_x + \delta_x)(\lambda_x + \delta_y) - (\lambda_x + \delta_x)a - (\Delta_x + \delta_x)b + c = (x + a)(y + b) - (y + b)a - (x + a)b + ab = xy = z\). Finally, to retain the knowledge of \(\Delta\)-values, parties compute \(\langle \Delta \rangle = \langle z \rangle + \langle \lambda_x \rangle\) and open it to get \(\Delta_x\).

#### 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 SPDZ\(_{2^k}\)-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 \(\langle z \rangle\), followed by computing \(\langle \Delta \rangle = \langle z \rangle + \langle \lambda_z \rangle\) and opening \(\Delta_x\). Patra et al. have noted in ABY2.0 [45] that it is unnecessary for the parties to compute and store the value \(\langle z \rangle\). Instead, only \(\langle \lambda_z \rangle\) is needed. This also holds for addition gates, where parties only need to compute \(\Delta_x = \lambda_x + \lambda_y\) locally, without computing and storing \(\langle z \rangle\). 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 \(\Delta\)-values associated with the output wire of that gate, based on the \(\Delta\)-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 SPDZ\(_{2^k}\) preprocessing functionality \(\mathcal{F}_{\text{Prep}}\) [9] for generating randomness. \(\mathcal{F}_{\text{Prep}}\) 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 \(\mathcal{F}_{\text{Prep}}\) is given in Appendix A.1.

We are now ready to present the arithmetic-circuit preprocessing protocol \(\Pi_{\text{PrepArith}}\). The goal of \(\Pi_{\text{PrepArith}}\) 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 \(\delta\)-values for each

---

1Note 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 $F_{\text{PrepArith}}$, which is described in Appendix A.2 due to space constraints. The protocol $\Pi_{\text{PrepArith}}$ is presented below.

### Protocol 1: $\Pi_{\text{PrepArith}}$

The parties proceed in topological order of the circuit.

**Input:** For each input wire $x$ of party $P_i$, parties call $F_{\text{Prep}}$. Input to get $(\lambda_x, \lambda_x)$ where $\lambda_x$ 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 $[\lambda] = [\lambda_x] + [\lambda_y]$.

**Multiply:** For a multiplication gate with input wires $x$, $y$ and output wire $z$, all the parties do the following:

1. Call $F_{\text{Prep}}$. 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 $F_{\text{Prep}}$.Rand to get $[\lambda_z]$, where $\lambda_z \in \mathbb{Z}_{2^k}$.

Now the parties have $[a], [b], [c], [\lambda_x], [\delta_x], [\delta_y]$ for a multiplication gate.

**Output:** Parties open the shared $[\delta]-$values for every multiplication gate to get them in the clear, then run $\Pi_{\text{MACCheck}}$ to check the MACs on values that are opened. If $\Pi_{\text{MACCheck}}$ aborts then parties abort. Otherwise the protocol completes without failure.

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

### Protocol 2: $\Pi_{\text{OnlineArith}}$

**Initialize:** Parties call $F_{\text{PrepArith}}$ with the circuit to get the $\delta$-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 $\Delta_i = x + \lambda_x$, then all parties store $\Delta_i$.

**Add:** For an addition gate with input wires $x$, $y$ and output wire $z$, parties locally compute $\Delta = \Delta_x + \Delta_y$.

**Multiply:** For a multiplication gate with input wires $x$, $y$ and output wire $z$, all parties do the following:

1. Locally compute $[\Delta] = ([\Delta_x + \delta_x]) ([\Delta_y + \delta_y]) - ([\Delta_x + \delta_x]) [a] - ([\Delta_y + \delta_y]) [b] + [c] + [\lambda_z]$.

2. Open $\Delta_z$ to get $\Delta_z$ in the clear.

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

1. Locally compute $[x] = [\Delta] - [\lambda_x]$ and open $x$ in the clear.
2. Run $\Pi_{\text{MACCheck}}$ on all the values that have been opened so far. If $\Pi_{\text{MACCheck}}$ 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 $\Pi_{\text{PrepArith}}$ securely implements the functionality $F_{\text{PrepArith}}$ against a static, malicious adversary corrupting up to $n - 1$ parties in the $F_{\text{Prep}}$-hybrid model.

**Theorem 2.** The protocol $\Pi_{\text{OnlineArith}}$ securely implements the functionality $F_{\text{OnlineArith}}$ against a static, malicious adversary corrupting up to $n - 1$ parties in the $F_{\text{PrepArith}}$-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 $\lambda$-values are the same as ours. However, instead of letting the parties have a multiplication triple $(a, b, c)$ (which are independent of the $\lambda$-values), ABY2.0 let the parties hold a sharing of $\lambda_x$, $\lambda_y$ (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 $\lambda_x, \lambda_y$.

This does not work for maliciously secure dishonest majority MPC, even if the underlying secret-sharing scheme of ABY2.0 is replaced by the SPDZ$^2$-sharing. The triple generation protocol in malicious setting [9] cannot take two determined values $(\lambda_x, \lambda_y)$ 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.

²Although 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 $\mathbb{F}$. 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 reactivity 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 $\mathbb{Z}_{2^k+1}$, 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 MPClearn [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 $\Pi_{\text{OnlineArith}}$, where $\Delta = z' + \lambda$, is opened, it is possible to locally truncate it to obtain $\Delta = \Delta/2^d$. Assuming the parties possess $[\lambda_c]$ such that $\lambda_c = \lambda_c/2^d$, it follows that $\delta = \delta - \lambda_c = \Delta/2^d - \lambda_c/2^d = 2^d', \therefore$ thereby completing the truncation. The crucial point is that $([\lambda_c], [\lambda_c])$ not only functions as random masks in $\Delta$, but also serves as a truncation pair.

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

---

**Procedure 1: $\pi_{\text{MultTrunc}}$**

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 $\mathcal{F}_{\text{Prep}}.\text{Triple}$ to get $([a], [b], [c])$.
2. Parties compute $[\delta_a] = [a] - [\lambda_a]$, $[\delta_b] = [b] - [\lambda_b]$ and open $\delta_c$, $\delta_c$.
3. Parties call $\mathcal{F}_{\text{edaBits}}$ on input $(k - d)$ to get $[\lambda_c]$ and its bit decomposition $\{(\lambda_c[i])_{i=0}^{k-d-1}\}$.
4. Parties call $\mathcal{F}_{\text{edaBits}}$ on input $d$ to get $[u]$ and its bit decomposition $\{(u[i])_{i=0}^{d-1}\}$.
5. Parties locally compute $[\lambda_c] = 2^d \cdot [\lambda_c] + [u]$.

**Online phase:**
Online communication cost. In the online phase of procedure \( \mathcal{P}_{\text{MultTrunc}} \), each party is only required to broadcast a single ring element. This communication cost is identical to that of multiplication in \( \Pi_{\text{OnlineArith}} \). 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 \( \mathcal{P}_{\text{MultTrunc}} \) we have the following proposition.

\( \textbf{Proposition 3. In the procedure } \mathcal{P}_{\text{MultTrunc}}, \text{ if } xy \leq 2^\ell \text{ for some } \ell < k, \text{ then with probability at least } 1 - 2^{-k - \ell}, \text{ it holds that } z = \cdot x \text{ satisfies} \)

\[
\sum_{i=1}^{m} \bar{a}[i] y[i] = \sum_{i=1}^{m} \bar{a}[i] y[i].
\]

A trivial way is to execute \( m \) multiplications in \( \Pi_{\text{OnlineArith}} \) 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 \([\lambda_c][i]\) for each multiplication in the dot product. Instead, only a single random share \([\lambda_c]\) is needed for the final result. Then in the online phase, parties locally compute and sum up the products before masking with \([\lambda_c]\) to get \([\Delta_c]\). Finally, parties open \( \Delta_c \) to retain knowledge on \( \Delta_c \). For details, see our dot product procedure \( \mathcal{P}_{\text{DotProduct}} \) below.

### Procedure 2: \( \mathcal{P}_{\text{DotProduct}} \)

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

**Preprocessing phase:**

1. Parties call \( \mathcal{J}_{\text{Prep}} \). Rand to get \([\lambda_c]\) where \( \lambda_c \sim \mathcal{Z}_{2^k} \).

2. Parties call \( \mathcal{J}_{\text{Prep}} \). 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 \( \mathcal{J}_{\text{Prep}} \).

3. Parties compute \([\vec{\delta}_1] = [\vec{a}] - [\vec{\lambda}_c], [\vec{\delta}_2] = [\vec{b}] - [\vec{\lambda}_c] \) and open \( \vec{\delta}_1, \vec{\delta}_2 \).

**Online phase:**

1. Parties locally compute \([\Delta_c] = \sum_{i=1}^{m} (\Delta_c[i] + \delta_1[i] + \delta_2[i] - \delta_1[i] - \delta_2[i])\).

2. Parties open \( \Delta_c \) to get \( \Delta_c \) in the clear.

### 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., \([\lambda]\) indicates that each element of the vector \( \lambda \) is \([\cdot]\)-shared.

For a dot product gate with input vectors \( \vec{x}, \vec{y} \) of length \( m \), the goal of the procedure \( \mathcal{P}_{\text{DotProduct}} \) is to compute the output on wire \( z \) where \( z = \vec{x} \cdot \vec{y} = \sum_{i=1}^{m} x[i] y[i] \).
The dot-product-then-truncate procedure $\Pi_{\text{DotProductTrunc}}$ can be constructed similarly as $\Pi_{\text{MultTrunc}}$. We describe the details of $\Pi_{\text{DotProductTrunc}}$ in Appendix B.2. In addition, matrix multiplication with truncation also benefits from $\Pi_{\text{DotProductTrunc}}$. 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 $\text{SPD}_2 + [15]$ 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 $\text{SPD}_2 + [15]$ 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 $F_{\text{edaBits}}$ to get a shared random mask $[r]$ and its bit decomposition $\{[r_i]_2\}_{i=0}^{k-1} = ([r_1]_2, \ldots, [r_k]_2)$. In the online phase, the parties compute $[c] = [x] + [r]$ and open $c$ in the clear, then use a bit-wise comparison circuit $\Pi_{\text{BltL}}$ to compare the public value $c$ and the secret-shared bits $\{[r_i]_2\}_{i=0}^{k-1}$, obtaining $[z]$, where $z = (c < r) = (x + r < r) = (x < 0)$. Finally, $[z]$ is converted to $[z]$ using a bit-to-arithmetic conversion functionality $F_{\text{B2A}}$. The procedure for standard bit-wise comparison $\Pi_{\text{BltL}}$ is described in Appendix B.3, and the bit-to-arithmetic conversion functionality $F_{\text{B2A}}$ is described in Appendix A.4.

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

To apply the mask-and-compare technique, the parties must open the value $[c] = [x] + [r] = \Delta_c - [\lambda_x] + [r]$. Let $\delta_c = r - \lambda_x$, observe that $[\delta_c] = [r] - [\lambda_x]$ can be computed and opened in the preprocessing phase. Consequently, $c = \Delta_c + (r - \lambda_x) = \Delta_c + \delta_c$ 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. For details, see the LTZ procedure $\Pi_{\text{LTZ}}$ 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 $\Pi_{\text{BltL}}$ 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 $\Pi_{\text{LTZ}}$, obtaining the shared bit decomposition of $\lambda_x$ requires an expensive protocol [5]. Hence, we resort to generating an additional random mask $[r]$ using the relatively cheap $F_{\text{edaBits}}$.

**Procedure 3: $\Pi_{\text{LTZ}}$**

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

**Preprocessing phase:**

1. Parties call $F_{\text{edaBits}}$ on input $k$ to get $[r]$ and its bit decomposition $\{[r_i]_2\}_{i=0}^{k-1} = ([r_1]_2, \ldots, [r_k]_2)$.
2. Parties compute $[\delta_c] = [r] - [\lambda_x]$, then open $\delta_c$.
3. Parties call $F_{\text{B2A}}$ on input $[z]_2$ to get $[z]$.
4. Parties locally compute $[\lambda_c] = [\lambda_x] + [z]$.

**Online phase:**

1. Parties call $\Pi_{\text{BltL}}(\Delta_c + [\lambda_x], [\delta_c]_2)_{i=0}^{k-1}$ to get $[z]_2$.
2. Parties call $F_{\text{B2A}}$ on input $[z]_2$ to get $[z]$.
3. Parties locally compute $[\Delta_c] = [\lambda_c] + [z]$.
4. Parties open $\Delta_c$ to get $\Delta_c$ in the clear.

**Further optimization for the preprocessing phase.** If procedure $\Pi_{\text{LTZ}}$ is to be executed right after procedure $\Pi_{\text{MultTrunc}}$, 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 $\Pi_{\text{MultTrunc}}$, 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 parties replace the first step with $\Pi_{\text{BltL}}(\Delta_c, [\lambda_x]_2, [\delta_c]_2)$, Correctness holds since $(x < 0) = (\Delta_c - \lambda_x < 0) = (\Delta_c < \lambda_x)$. As a result, the call to $F_{\text{edaBits}}$ and the opening of $\delta_c$ 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 $\Pi_{\text{LTZ}}$ 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 $F_{\text{B2A}}$), and an invocation of $\Pi_{\text{BltL}}$ on length-$\lambda$ 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 $\Pi_{PrepPPML}$, and an online protocol $\Pi_{OnlinePPML}$. These are put in Appendix C due to space constraints, and we briefly describe them below.

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

We define the corresponding functionalities $F_{PrepPPML}$ and $F_{OnlinePPML}$, 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 $\Pi_{PrepPPML}$ securely implements the functionality $F_{PrepPPML}$ against a static, malicious adversary corrupting up to $n - 1$ parties in the $(F_{Prep}, F_{edaBits})$-hybrid model.

Theorem 5. The protocol $\Pi_{OnlinePPML}$ securely implements the functionality $F_{OnlinePPML}$ against a static, malicious adversary corrupting up to $n - 1$ parties in the $(F_{Prep}, F_{B2A})$-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 $f : \mathbb{R}^n \rightarrow \mathbb{Z}_q$ defined as

$$f(\vec{x}) = \arg \max_{i \in \{1,...,q\}} (W[i] \cdot \vec{x} + \vec{b}[i])$$

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 $W \in \mathbb{R}^{q \times n}$ and the vector $\vec{b} \in \mathbb{R}^n$ are the parameters. Note that $W[i]$ denotes the i-th row of $W$.

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

Activation functions. We mainly consider the ReLU function $ReLU(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$. 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 $ReLU(x) = (x > 0) \cdot x = (1 - (x < 0)) \cdot x$, parties can invoke $\pi_{LTZ}$ 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++. To compare with the state of the art SPDZ$_{2+}$ [15], we implement the evaluations for SPDZ$_{2+}$ in the MP-SPDZ framework [23]. Although both SPDZ$_{2+}$ 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 SPDZ$_{2+}$ secret-sharing scheme [9] in the evaluations of both MD-ML and SPDZ$_{2+}$ [15]. For fixed point arithmetic in MD-ML, we use $d = 20$ for the number of fractional bits.

---

3Available 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 parties (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$\mathbb{Z}_{2k}+$ 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$\mathbb{Z}_{2k}+$:

- 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 $4.0-20.0 \times$ faster in LAN, $6.0-73.5 \times$ faster in WAN, and $1.7-8.8 \times$ more communication-efficient compared to SPD$\mathbb{Z}_{2k}+$.

6.2.2 Benchmarks of the preprocessing phase

We benchmark the preprocessing phase of the following building blocks for both MD-ML and SPD$\mathbb{Z}_{2k}+$ (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 $\pi_{\text{DotProduct}}$, $\pi_{\text{MultTrunc}}$, and $\pi_{\text{LTZ}}$ 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 $\pi_{\text{MultTrunc}}$ and $\pi_{\text{LTZ}}$ are nearly the same as SPD$\mathbb{Z}_{2k}+$, while the preprocessing cost of our vector dot product procedure is slightly higher than SPD$\mathbb{Z}_{2k}+$. The reason is that parties need to generate the $[\lambda]$-values and open the $\delta$-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:

- SVM inference on the MNIST, CIFAR-10, and Tiny ImageNet datasets with both MD-ML and SPD$\mathbb{Z}_{2k}+$ (Table 8).
- LeNet inference on the MNIST, CIFAR-10, and Tiny ImageNet datasets with both MD-ML and SPD$\mathbb{Z}_{2k}+$ (Table 9).
- AlexNet inference on the CIFAR-10, Tiny ImageNet, and ImageNet datasets with both MD-ML and SPD$\mathbb{Z}_{2k}+$ (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$\mathbb{Z}_{2k}+$ is observed from the tables. For LeNet inference, we are $3.4-9.2 \times$ faster in LAN, $9.7-31.7 \times$ faster in WAN, and $2.0-2.9 \times$ more communication-efficient. For AlexNet inference, we are $8.0-11.0 \times$ faster in LAN, $93.3-157.7 \times$ faster in WAN, and $9.8-23.8 \times$ more communication-efficient. For ResNet-18 inference on CIFAR-10, we are capable of completing the online phase of inference in $25.8$ in LAN and $363.9$ 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 $0.13\% - 0.34\%$ compared to plaintext computation.

---

Footnote: 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

<table>
<thead>
<tr>
<th>Operations</th>
<th>LAN Time</th>
<th>WAN Time</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ours</td>
<td>SPD(Z_{k^2})</td>
<td>Factor</td>
</tr>
<tr>
<td>Dot product</td>
<td>2.5 ms</td>
<td>46.8 ms</td>
<td>18.7×</td>
</tr>
<tr>
<td>LTZ</td>
<td>278 ms</td>
<td>1271 ms</td>
<td>4.57×</td>
</tr>
</tbody>
</table>

†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

<table>
<thead>
<tr>
<th>Shape</th>
<th>LAN Time</th>
<th>WAN Time</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ours</td>
<td>SPD(Z_{k^2})</td>
<td>Factor</td>
</tr>
<tr>
<td>1000, 2048, 1</td>
<td>35.3 ms</td>
<td>717 ms</td>
<td>20.0×</td>
</tr>
<tr>
<td>32, 32, 32</td>
<td>0.77 ms</td>
<td>9.05 ms</td>
<td>11.8×</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>(HW,C,M,h,s)</th>
<th>LAN Time</th>
<th>WAN Time</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ours</td>
<td>SPD(Z_{k^2})</td>
<td>Factor</td>
</tr>
<tr>
<td>28×28, 1, 1, 3, 1</td>
<td>1 ms</td>
<td>7.2 ms</td>
<td>7.2×</td>
</tr>
<tr>
<td>32×32, 3, 1, 3, 1</td>
<td>2.15 ms</td>
<td>8.6 ms</td>
<td>4.0×</td>
</tr>
</tbody>
</table>

† \(HW\) 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

<table>
<thead>
<tr>
<th>Operation</th>
<th>LAN Time</th>
<th>WAN Time</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ours</td>
<td>SPD(Z_{k^2})</td>
<td>Factor</td>
</tr>
<tr>
<td>MultTrunc</td>
<td>2.191 s</td>
<td>2.189 s</td>
<td>0.999×</td>
</tr>
<tr>
<td>LTZ</td>
<td>2.388 s</td>
<td>2.390 s</td>
<td>1.001×</td>
</tr>
<tr>
<td>Dot prod.</td>
<td>8.065 s</td>
<td>6.246 s</td>
<td>0.775×</td>
</tr>
</tbody>
</table>

†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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>LAN Time</th>
<th>WAN Time</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ours</td>
<td>SPD(Z_{k^2})</td>
<td>Factor</td>
</tr>
<tr>
<td>MNIST</td>
<td>0.92 ms</td>
<td>4 ms</td>
<td>4.3×</td>
</tr>
<tr>
<td>CIFAR-10</td>
<td>2.2 ms</td>
<td>21 ms</td>
<td>9.5×</td>
</tr>
<tr>
<td>Tiny ImageNet</td>
<td>47 ms</td>
<td>600 ms</td>
<td>12.8×</td>
</tr>
</tbody>
</table>
Table 9: Online communication and running time of LeNet inference on different datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Ours LAN Time</th>
<th>SPD(\mathbb{Z}_{2^k}) + Factor</th>
<th>Ours WAN Time</th>
<th>SPD(\mathbb{Z}_{2^k}) + Factor</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>80 ms</td>
<td>274 ms</td>
<td>3.4×</td>
<td>9.82 s</td>
<td>95.647 s</td>
</tr>
<tr>
<td>CIFAR-10</td>
<td>89 ms</td>
<td>399 ms</td>
<td>4.5×</td>
<td>10.14 s</td>
<td>117.309 s</td>
</tr>
<tr>
<td>Tiny ImageNet</td>
<td>196 ms</td>
<td>1798 ms</td>
<td>9.2×</td>
<td>13.968 s</td>
<td>443.397 s</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Ours LAN Time</th>
<th>SPD(\mathbb{Z}_{2^k}) + Factor</th>
<th>Ours WAN Time</th>
<th>SPD(\mathbb{Z}_{2^k}) + Factor</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIFAR-10</td>
<td>0.82 s</td>
<td>6.80 s</td>
<td>8.3×</td>
<td>34.88 s</td>
<td>3254.7 s</td>
</tr>
<tr>
<td>Tiny ImageNet</td>
<td>2.06 s</td>
<td>16.40 s</td>
<td>8.0×</td>
<td>53.89 s</td>
<td>6774.6 s</td>
</tr>
<tr>
<td>ImageNet</td>
<td>7.38 s</td>
<td>81.35 s</td>
<td>11.o×</td>
<td>188.92 s</td>
<td>29785.2 s</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Model and Dataset</th>
<th>LAN</th>
<th>WAN</th>
<th>Comm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet-18 on CIFAR-10</td>
<td>25.8 s</td>
<td>362.9 s</td>
<td>4.15 GB</td>
</tr>
</tbody>
</table>

Table 12: Inference Accuracy on MNIST dataset

<table>
<thead>
<tr>
<th>Model</th>
<th>Plaintext</th>
<th>MD-ML</th>
<th>Relative Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM</td>
<td>94.42%</td>
<td>94.28%</td>
<td>0.153%</td>
</tr>
<tr>
<td>LeNet</td>
<td>98.86%</td>
<td>98.43%</td>
<td>0.435%</td>
</tr>
</tbody>
</table>

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


A Functionalities

A.1 SPD\(Z_{2^k}\) Preprocessing Functionality

Functionality \(\mathcal{F}_{\text{Prep}}\) is taken from \cite{Prep} and is described below.

**Functionality 1: \(\mathcal{F}_{\text{Prep}}\)**

**Input:** On input \((\text{Input},P_i)\) from all parties, sample random \(r \triangleq \mathbb{Z}_{2^k}\) 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 \((\text{Triple})\) from all parties, sample two random values \(a,b \triangleq \mathbb{Z}_{2^k}\), compute \(c = ab \mod 2^k\), generate \([a],[b],[c]\), and distributes the sharings \([a],[b],[c]\) to all parties.

**Rand:** On input \((\text{Rand})\) from all parties, sample random value \(r \triangleq \mathbb{Z}_{2^k}\), 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: \(\mathcal{F}_{\text{PrepArith}}\)**

**Input:** On input \((\text{Input},P_i,id)\) from the parties, where \(id\) is a fresh, valid identifier for the input wire of \(P_i\), the functionality samples \(\lambda_x \triangleq \mathbb{Z}_{2^k}\) 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 \([\lambda_x]\) to \(P_i\) and distributes \([\lambda_x]\) to all parties.

**Add:** On input \((\text{Add},id_1,id_2,id_3)\) from the parties, the functionality retrieves (if present in memory) the values \((id_1,[\lambda_1]),(id_2,[\lambda_2])\) and stores \((id_3,[\lambda_3])\) where \(\lambda_z = \lambda_x + \lambda_y\).

**Multiply:** On input \((\text{Multiply},id_1,id_2,id_3)\) from the parties, the functionality retrieves (if present in memory) the values \((id_1,[\lambda_1]),(id_2,[\lambda_2])\). The functionality then samples \(a,b,\lambda_z \triangleq \mathbb{Z}_{2^k}\), computes \(c = ab\), generates \([a],[b],[c],[\lambda_z]\), and distributes them to the parties. Finally, the functionality computes \(\delta_a = a - \lambda_x, \delta_b = b - \lambda_y\) and stores \((id_3,[\lambda_3])\) and \(\delta_a,\delta_b\).

**Output:** The functionality sends all the stored \(\delta\)-values to the adversary, then waits for a message Abort or Proceed from the adversary: if it sends Abort then the functionality aborts, otherwise the functionality sends the \(\delta\)-values to the parties.
A.3 Our Functionalities for PPML

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

**Functionality 3: \( F_{\text{OnlineArith}} \)**

Initialize: Receive (init, k) from all parties.
Input: On input (Input, \( \mathcal{P}_i, \text{id}_x \)) from \( \mathcal{P}_i \) and input (Input, \( \mathcal{P}_j \)) from other parties, where \( \text{id} \) is a fresh identifier, store \( (\text{id}, x) \).
Add: On input (Add, \( \text{id}_1, \text{id}_2, \text{id}_3 \)) from all the parties (where \( \text{id}_1 \) and \( \text{id}_2 \) are present in memory), retrieve \( (\text{id}_1, x), (\text{id}_2, y) \) and store \( (\text{id}_3, x+y) \).
Multitype: On input (Mult, \( \text{id}_1, \text{id}_2, \text{id}_3 \)) from all the parties (where \( \text{id}_1, \text{id}_2 \) are present in memory), retrieve \( (\text{id}_1, x), (\text{id}_2, y) \) and store \( (\text{id}_3, xy) \).
Output: On input (Output, \( \text{id} \)) from all the parties (where \( \text{id} \) is present in memory), retrieve \( (\text{id}, y) \) and output it to the adversary. Wait for an input Proceed or Abert from the adversary. If this is Proceed then send \( y \) to all parties, otherwise abort.

**Functionality 4: \( F_{\text{PrepPPML}} \)**

Functionality \( F_{\text{PrepPPML}} \) has the same commands as \( F_{\text{OnlineArith}} \), in addition to the following:

- **MultTrunc**: On input (MultTrunc, \( \text{id}_1, \text{id}_2, \text{id}_3 \)) from all parties (where \( \text{id}_1, \text{id}_2 \) are present in memory), retrieve \( (\text{id}_1, x), (\text{id}_2, y) \) and store \( (\text{id}_3, xy/2^d) \).

**Functionality 5: \( F_{\text{OnlinePPML}} \)**

Functionality \( F_{\text{OnlinePPML}} \) has the same commands as \( F_{\text{OnlineArith}} \), in addition to the following:

- **MultTrunc**: On input (MultTrunc, \( \text{id}_1, \text{id}_2, \text{id}_3 \)) from all parties (where \( \text{id}_1, \text{id}_2 \) are present in memory), retrieve \( (\text{id}_1, x), (\text{id}_2, y) \) and store \( (\text{id}_3, xy/2^d) \).

**Functionality 6: \( F_{\text{edaBits}} \)**

On input (edaBits, \( \ell \)) from all parties, sample uniformly random values \( (r_1, \ldots, r_{\ell-1}) \in \mathbb{Z}_2 \), compute \( r = \sum_{i=0}^{\ell-1} 2^i r_i \), generate \( \{r_i\}_{i=1}^{\ell-1} \) and \( \lceil \rceil \), then distributes the sharings to all parties.

**Functionality 7: \( F_{\text{B2A}} \)**

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

**A.4 Functionalities for edaBits and B2A**

Functionality \( F_{\text{edaBits}} \) is taken from [17].

**Functionality \( F_{\text{B2A}} \)**

Functionality \( F_{\text{B2A}} \) is a standard conversion functionality.

B Procedures

B.1 MAC Checking

The procedure \( \pi_{\text{MACCheck}} \) is taken from [9]. It relies on a coin tossing functionality \( F_{\text{Rand}} \) to sample public random values for the parties.

**Procedure 4: \( \pi_{\text{MACCheck}} \)**

Procedure \( \pi_{\text{MACCheck}} \) takes as input a set of opened values \( (x_1, x_2, \ldots, x_t) \) and checks the correctness of the corresponding MACs. Let \( x_1^i, m_1^i, a_i^i \) be \( P_j \)’s share, MAC share and MAC key share for \( x_i \).

1. Parties call \( F_{\text{Rand}} \) to sample public random values \( (r_1, r_2, \ldots, r_t) \in \mathbb{Z}_2^t \), and compute \( y = \sum_{i=1}^{t} r_i \cdot x_i \mod 2^{k_x+y} \).

2. Each Party \( P_j \) computes \( m_1^i = \sum_{i=1}^{t} r_i m_i^i \) and \( z_1^i = m_1^i - a_i^i \cdot y \), then it commits to \( z_1^i \).
Theorem 6 ([9, Theorem 1]). Assume $\alpha$ is uniformly random to the adversary, if procedure $\pi_{MACCheck}$ passes, then the values accepted by the parties are correct, except with probability at most $2^{-\alpha + \log(n+1)}$.

B.2 Dot Product with Truncation

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

Procedure 5: $\pi_{DotProductTrunc}$

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 $F_{\text{Prep}}$. Triple $m$ times to get $m$ triples, obtaining $[\vec{a}], [\vec{b}], [\vec{c}]$, where $([\vec{a}],[\vec{b}],[\vec{c}])$ is an individual multiplication triple obtained in the $i$-th call of $F_{\text{Prep}}$.
2. Parties compute $[\vec{\delta}_a] = [\vec{a}] - [\vec{\lambda}_a], [\vec{\delta}_b] = [\vec{b}] - [\vec{\lambda}_b]$ and open $\vec{\delta}_a, \vec{\delta}_b$.
3. Parties call $F_{\text{edabits}}$ on input $(k-d)$ to get $[\vec{\lambda}_c]$ and its bit decomposition $\{[\vec{\lambda}_c]_i\}_i=0^{d-1}$.
4. Parties call $F_{\text{edabits}}$ on input $d$ to get $[\vec{a}]$ and its bit decomposition $\{[\vec{a}]_i\}_i=0^{d-1}$.
5. Parties locally compute $[\vec{\lambda}_c] = 2^d \cdot [\vec{\lambda}_c] + [\vec{a}]$.

**Online phase:**

1. Parties locally compute $\Delta_c = \sum_{i=1}^{m} \left( (\vec{\Delta}_c[i] + \vec{\delta}_a[i] \vec{\Delta}_a[i] + \vec{\delta}_b[i] \vec{\Delta}_b[i]) - (\vec{\Delta}_a[i] + \vec{\delta}_b[i] \vec{\Delta}_b[i]) \vec{\delta}_a[i] - (\vec{\Delta}_a[i] + \vec{\delta}_b[i] \vec{\Delta}_b[i] \vec{\delta}_a[i]) + [\vec{\lambda}_c] \right)$.
2. Parties open $\Delta_c$ to get $\Delta_c$ in the clear.
3. Parties locally compute $\Delta_c = \Delta_c / 2^d$.

B.3 Bit-wise Comparison

The bit-wise comparison procedure $\pi_{\text{BitLT}}$ is taken from [15]. The procedure uses a circuit $\pi_{\text{Carry}}(x, \{y_i\}_i=0, u)$ to compute the carry bit of an addition between two integers $x, y \in \mathbb{Z}_{2k}$, when the initial carry-in bit is $u \in \{0,1\}$.

The $\pi_{\text{BitLT}}$ procedure is described below, we denote it as $\pi_{\text{BitLT}}(x, \{y_i\}_i=0^k, 1)$.

C Protocols

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

Protocol 3: $P_{\text{PrepPPML}}$

The parties proceed in topological order.

**Input:** Same as in $P_{\text{PrepArith}}$.

**Add:** Same as in $P_{\text{PrepArith}}$.

**Multiply:** Same as in $P_{\text{PrepArith}}$.

**MultTrunc:** Parties execute the preprocessing phase of $P_{\text{MultTrunc}}$, the opening of the $\delta$-values can be deferred and batched in the Output phase.

**DotProduct:** Parties execute the preprocessing phase of $P_{\text{DotProduct}}$, the opening of the $\delta$-values can be deferred and batched in the Output phase.

**LTZ:** Parties execute the preprocessing phase of $P_{\text{LTZ}}$, the opening of the $\delta$-values can be deferred and batched in the Output phase.

**Output:** Same as in $P_{\text{PrepArith}}$.

Protocol 4: $P_{\text{OnlinePPML}}$

**Initialize:** Parties call $F_{\text{PrepPPML}}$ with the circuit to get the $\delta$-values, the shared $[\vec{\lambda}]$-values, and the multiplication triples for each gate.

Then, parties proceed in topological order.

**Input:** Same as in $P_{\text{OnlineArith}}$.

**Add:** Same as in $P_{\text{OnlineArith}}$.

**Multiply:** Same as in $P_{\text{OnlineArith}}$.

**MultTrunc:** Parties execute the online phase of $P_{\text{MultTrunc}}$.

**DotProduct:** Parties execute the online phase of $P_{\text{DotProduct}}$.

**LTZ:** Parties execute the online phase of $P_{\text{LTZ}}$.

**Output:** Same as in $P_{\text{OnlineArith}}$. 