SoK: Truncation Untangled: Scaling Fixed-Point Arithmetic for Privacy-Preserving Machine Learning to Large Models and Datasets

Christopher Harth-Kitzerow
Technical University of Munich, BMW Group
christopher.harth-kitzerow@tum.de

Ajith Suresh
Technology Innovation Institute, Abu Dhabi
ajith.suresh@tii.ae

Georg Carle
Technical University of Munich
carle@net.in.tum.de

Abstract

Fixed Point Arithmetic (FPA) is widely used in Privacy-Preserving Machine Learning (PPML) to efficiently handle decimal values. However, repeated multiplications in FPA can lead to overflow, as the fractional part doubles in size with each multiplication. To address this, truncation is applied post-multiplication to maintain precision. Various truncation schemes based on Secure Multiparty Computation (MPC) exist, but trade-offs between accuracy and efficiency in PPML models and datasets remain underexplored. In this work, we analyze and consolidate different truncation approaches from the MPC literature.

We conduct the first large-scale systematic evaluation of PPML inference accuracy across truncation schemes, ring sizes, neural network architectures, and datasets. Our study provides clear guidelines for selecting the optimal truncation scheme and parameters for PPML inference. All evaluations are implemented in the open-source HPMPC MPC framework [1] , facilitating future research and adoption. Beyond our large scale evaluation, we also present improved constructions for each truncation scheme, achieving up to a fourfold reduction in communication and round complexity over existing schemes. Additionally, we introduce optimizations tailored for PPML, such as strategically fusing different neural network layers. This leads to a mixed-truncation scheme that balances truncation costs with accuracy, eliminating communication overhead in the online phase while matching the accuracy of plaintext floating-point PyTorch inference for VGG-16 on the ImageNet dataset.

Keywords

Fixed-point arithmetic, MPC, PPML, Truncation, Secure Inference

1 Introduction

Privacy-Preserving Machine Learning (PPML) [32, 35] aims to enable machine learning model training and inference while keeping model parameters and data private using cryptographic techniques. Secure Multiparty Computation (MPC) [29] allows multiple parties to jointly compute a function on their private inputs without revealing any information about the inputs to others, making it a powerful building block for efficient PPML [35].

While private training of state-of-the-art (SOTA) neural networks is advancing in reducing MPC's performance overhead [22, 47], private inference using MPC is already practical for many models and datasets. For example, the recent PIGEON [18] PPML framework demonstrated for the first time that secure inference of various SOTA convolutional neural networks (CNNs) on the popular ImageNet image classification dataset achieves a throughput of over 10 images per second. These advancements highlight the need for a systematic study of different MPC-specific configurations and their impact on the performance and accuracy of large CNN models such as VGG-16 [43] and various ResNet [19] architectures.

Plaintext Machine Learning typically utilizes floating-point numbers to represent decimal values. However, SOTA libraries for floating-point arithmetic in MPC [39] introduce significant communication overhead compared to integer arithmetic, as even floating-point addition requires an interactive protocol. To efficiently compute on secret-shared decimal numbers, MPC-based PPML algorithms rely on fixed-point arithmetic (FPA) [7] techniques. FPA encodes decimal values with fixed precision as a ring element over an ℓ-bit ring $\mathbb{Z}_{2^\ell}$, ensuring it accommodates the expected input range. This enables MPC parties to perform integer-only arithmetic using these encoded decimal values.

In FPA, decimal values are represented as ℓ-bit integers, with k bits for the fractional part and $\ell - k$ bits for the integer part. Multiplying two fixed-point numbers results in $2k$ fractional bits and $\ell - 2k$ integer bits. Thus, repeated multiplications quickly lead to an overflow of integer bits, resulting in incorrect computation. To prevent overflow and maintain precision, the number of fractional bits must be reduced back to k using *truncation*. This process, which is equivalent to performing an arithmetic right shift by k bits, ensures the result has the correct number of integer and fractional bits after multiplying two FPA values.

While truncation is straightforward in plaintext computations, it is a non-trivial cryptographic operation in MPC. However, its communication overhead remains relatively low compared to floating-point arithmetic [24]. One drawback of using FPA in PPML is that it may not achieve the same level of accuracy as plaintext floating-point inference. Therefore, it is essential to carefully choose the fixed-point precision and bit width to minimize accuracy loss.

In recent years, various truncation methods have been proposed for different use cases and settings [10, 15, 33, 34]. These methods involve significant trade-offs between communication complexity and error probability. Thus, choosing one scheme over another can significantly impact both the accuracy and the overall runtime of PPML applications. This is especially important in layers such as

[1] Our implementation is integrated into HPMPC: https://github.com/chart21/hpmpc/

Proceedings on Privacy Enhancing Technologies 2025(4), 369–391

https://doi.org/10.56553/popets-2025-0135

average pooling, batch normalization, and linear layers, where truncation often accounts for most of the communication overhead. The trade-offs become more complex when comparing schemes across different ring sizes, since larger ring sizes can improve accuracy but at the cost of higher runtime.

Despite the availability of various truncation schemes and FPA configurations, such as different ring sizes and numbers of fractional bits, a comparison of approaches is lacking. In this work, we address this gap by systematically evaluating truncation techniques in PPML inference. Specifically, we focus on truncation for MPC protocols operating over the ring $\mathbb{Z}_{2^\ell}$, where $\ell \in \{16, 32, 64\}$, as these ring sizes align with native integer operations on modern hardware. While truncation is straightforward for field-based protocols due to the availability of division, field-based computations incur significant real-world overhead because modern hardware lacks native support for field operations. To provide a comprehensive comparison, we evaluate both runtime and accuracy across various truncation schemes, ring sizes, and a full range of practical fractional bit settings.

1.1 Related Work

While it is possible to perform floating-point operations in MPC [1, 13], SOTA floating-point protocols introduce significantly higher runtime overhead compared to fixed-point MPC. For instance, the SOTA floating-point addition protocol [39] requires 49 rounds of communication between parties and uses costly MPC primitives such as comparisons. In contrast, fixed-point addition can be performed by simply adding shares locally. In fact, both FPA addition and multiplication in MPC can use the same protocols as their integer counterparts. The only difference is that FPA multiplication requires an additional truncation step to maintain the correct bit-width. Therefore, developing efficient truncation primitives is essential for improving the performance of FPA-based MPC.

This section presents a brief overview of various truncation schemes proposed in the literature. The study of truncation in MPC using FPA semantics dates back to the work of Catrina et al. [6, 7]. However, in this work, we focus specifically on truncation in the context of MPC-based PPML algorithms. Additional technical details on the works covered in this section are provided in §3.

1.1.1 *Stochastic Truncation - $TS_{\{L\}}$.*

In this probabilistic approach proposed by SecureML [34], secret shares are locally truncated to obtain shares corresponding to the truncated value. SecureML operates in a two-party (2PC) semi-honest setting, where the value reconstructed from locally truncated shares deviates from the actual truncated value by at most 1 with very high probability. This construction was later generalized by ABY3 [33] for the three-party (3PC) setting. Since then, several works have proposed variants of this approach in 3PC [17, 38], 4PC [9, 11, 17, 25, 27], and general n-party [26] settings.

$TS_{\{L\}}$ can cause truncation failure, i.e. the truncation may introduce a large error with a certain probability. This probability increases with the absolute plaintext value's closeness to the ring size 2^ℓ [34, 51]. To mitigate this, frameworks employing such truncation schemes typically increase the ring size by an additional margin, commonly referred to as *slack*, reducing the probability of truncation failure.

Additionally, several works have explored fusing multiplication with truncation to reduce communication and round complexity [9, 33, 44]. For instance, the 3PC protocols in [17, 44] and the 4PC protocols in [9, 17, 27, 44] incur no additional communication or rounds when integrating multiplication with truncation. However, stand-alone truncations or multiplications with a public fixed-point value still require communication in these protocols.

Security of Stochastic Truncation. Li et al. [28] raised concerns that stochastic truncation schemes are inherently insecure under standard security definitions of MPC [5], since the truncation output depends on the same randomness that masks the input share. This finding motivated Orca [22] to propose a stochastic truncation scheme that does not rely on the same randomness as the input share. However, Santos et al. [42] showed that by using an alternative ideal functionality for stochastic truncation, all previously proposed truncation schemes can, in fact, be proven secure.

1.1.2 *Stochastic Truncation with Reduced Slack - $TS_{\{1\}}$.*

Dalskov et al. [10] proposed a stochastic truncation scheme that requires no slack but only guarantees correctness for positive plaintext values. They also provided an efficient construction for their scheme in the semi-honest 3PC setting. Fantastic Four [11] extended this approach to the malicious 4PC setting. Escudero et al. [15] introduced a simple modification that allows Dalskov et al.'s scheme to support negative values but requires a slack of one bit.

1.1.3 *Exact Truncation - $TE_{\{0\}}$.*

Exact truncation approaches [15] are independent of the value or the randomness of its secret shares and are equivalent to an arithmetic right shift in the plaintext domain. These approaches often involve share conversion between the arithmetic and Boolean domains. While exact truncation requires no slack, they introduce significant communication overhead due to the need for Boolean circuit computations, such as sign bit extraction or the addition of decomposed shares. Boolean adders can be implemented using Ripple Carry Adders (RCAs) or Parallel Prefix Adders (PPAs) [33], utilizing standard AND gates, multi-input AND gates [37], or multi-input scalar products [4]. Each approach presents trade-offs in communication rounds and message complexity, but all require at least $O(log(\ell))$ communication rounds and $O(\ell)$ messages exchanged in the Boolean domain.

1.1.4 *Exact Truncation with Slack - $TE_{\{1\}}$.*

To reduce the high costs of exact truncation, later works introduced a variant requiring a small slack of 1 bit while significantly lowering communication and computational complexity. Escudero et al. [15] proposed a generic construction in this setting that involves computing only two bit extraction circuits sequentially. Fantastic Four [11] presented a construction for the malicious 4PC setting, utilizing a most-significant bit extraction circuit and a t-least-significant bit extraction circuit in parallel, thereby reducing communication rounds.

Comparison of Truncation Schemes. While several works have proposed efficient ring-based truncation techniques, a comprehensive comparison of these techniques is largely missing, with a few exceptions. Piranha [49] analyzed the impact of different numbers of fractional bits on PPML inference accuracy across 2PC, 3PC, and 4PC settings. Based on their results, they recommended a ring size of 64 bits with 26 bits allocated for the fractional part when using

$TS_{\{L\}}$. However, Bicoptor 2.0 [51] found that Piranha did not sample random values in their experiments, which concealed the impact of truncation failure. Their experiments suggested that 15 fractional bits are more appropriate for a 64-bit ring to prevent truncation failure. This aligns with the range of 12 to 16 fractional bits commonly used in several works, such as ABY3 [33] and SWIFT [25].

Fantastic Four [11] empirically compared their stochastic truncation scheme in malicious 4PC setting, which requires only one bit of slack ($TS_{\{1\}}$), to the stochastic truncation scheme that requires a larger slack ($TS_{\{L\}}$). Their findings showed that, in a 64-bit ring, $TS_{\{1\}}$ was twice as efficient as $TS_{\{L\}}$. This efficiency gain was primarily due to the computational inefficiency of $TS_{\{L\}}$, which required 80-bit computations on 64-bit hardware to achieve the same truncation failure probability as $TS_{\{1\}}$.

Beyond the four truncation categories discussed earlier, some studies have introduced custom strategies tailored to their specific settings. For example, Cheetah [21] optimizes communication costs for exact truncation in the 2PC semi-honest setting by allowing a small error. In the 3PC semi-honest setting, Bicoptor 2.0 [51] proposes a truncate-then-multiply approach, where factors are truncated before multiplication. This method reduces the probability of truncation failure. However, a drawback of their approach is that it requires two truncations per multiplication instead of one. The recent work of MaSTer [50] improves the stochastic truncation $TS_{\{L\}}$ in ABY3 [33] for the 3PC malicious setting by carefully organizing the shares and incorporating a post-processing consistency check.

1.2 Research Gaps

Although various truncation schemes have been studied in MPC-based PPML inference, the trade-offs between accuracy and efficiency across different models and datasets remain largely underexplored. Similarly, strategies to improve the accuracy and efficiency of truncation primitives for specific MPC protocols or workloads, such as PPML, are lacking. In particular, we identify the following unanswered research questions (RQs) by the current literature:

RQ1: How do different stochastic and exact truncation approaches compare regarding slack requirements, communication complexity, and PPML inference accuracy?

RQ2: Which ring sizes and number of fractional bits should practitioners choose to achieve high PPML inference accuracy on various model architectures?

RQ3: Are there ways to reduce the communication overhead of existing truncation primitives when applied to SOTA MPC protocols?

RQ4: Are there optimizations specific to PPML applications to reduce FPA communication complexity or increase accuracy?

1.3 Our Contributions

In this work, we take a significant step towards bridging the identified gaps by analyzing and consolidating multiple truncation approaches from the MPC literature.

To answer RQ1 and RQ2, our study focuses on key aspects such as the impact of ring sizes, the relationship between truncation failure probabilities and slack sizes, and the accuracy implications of probabilistic truncation (cf. $TS_{\{L\}}$ in §1.1.1) compared to other approaches and plaintext floating-point inference. By systematically evaluating these aspects, we provide a comprehensive analysis of the trade-offs involved and present the following contributions:

Analytical Overview of Truncation Approaches (§3). We provide a systematic review of existing truncation approaches in MPC for privacy-preserving machine learning, building on the discussion in §1.1. For each of the four schemes—$TS_{\{L\}}$, $TS_{\{1\}}$, $TE_{\{0\}}$, and $TE_{\{1\}}$—analyzed in this work, we formally present their details and examine their corresponding slack requirements.

Systematic Evaluation (§6). We present the first comprehensive evaluation of PPML inference accuracy across various truncation schemes, ring sizes, neural network architectures, and datasets.

We implement all studied truncation approaches into the open-source HPMPC framework [16] and evaluate their runtime and accuracy across different ring sizes and neural network architectures using various benchmark datasets. Notably, we present the first evaluation of PPML inference accuracy on the ImageNet dataset [41], addressing the long-standing question of whether fixed-point MPC can scale to large-scale models and datasets [35].

We investigate how choices made during plaintext training influence truncation failure probabilities in PPML inference. Specifically, we find that training models with the ADAMW optimizer [31] benefits stochastic truncation due to its built-in weight decay mechanism, which helps reducing required slack sizes. Based on our extensive evaluation, we provide end-to-end guidelines on regularization techniques for plaintext training, optimal ring sizes and fractional bit lengths for inference, and the truncation approach that offers the best trade-off between communication complexity and accuracy.

To answer RQ3 and RQ4, we present several optimizations that improve the communication complexity and accuracy of truncation schemes.

Efficient Truncation Protocols (§4). We present efficient constructions for the truncation approaches studied in this work, tailored for the semi-honest 3PC and malicious 4PC honest-majority settings. These constructions are based on the SOTA Trio and Quad MPC protocols (PETS25) [17]. We observe that the novel sharing semantics of Trio and Quad can be utilized to replace several communication-intensive steps required by existing truncation primitives with local preprocessing or more efficient sub-protocols. Our approach reduces the communication overhead of existing truncation schemes by leveraging the unique sharing semantics of Trio and Quad. As shown in Table 1, our truncation primitives achieve up to a 4× improvement in communication complexity compared to the current SOTA methods.

Optimized Truncation for PPML (§5). In the context of PPML, we observe that truncation can be efficiently integrated with the evaluation of other layers by leveraging the inherent structure of neural networks. Specifically, computation and communication required by a layer's evaluation can often be merged with computation and communication required by a truncation primitive to obtain a fused primitive requiring a reduced number of messages and communication rounds compared to evaluating operations independently. This integration significantly reduces —and in some cases, entirely eliminates — the communication overhead associated with truncation. Furthermore, we propose optimizations to minimize the required slack size for truncation. For example, in

Table 1: Costs for truncating a secret-shared value by t bits. Notations: Pre. - preprocessing, On. - online, Rounds - communication rounds.

Approach	Setting	Protocol	Pre.	On.	Rounds
$TS_{\{L\}}$: Stochastic Truncation, Large Slack	3PC	ABY3 [33]	0	ℓ	1
		This Work	ℓ	**0**	**0**
		This Work[p]	0	0	0
	4PC	Tetrad [27]	ℓ	ℓ	0[c]
		This Work	ℓ	ℓ	0[c]
		Tetrad [27][p]	0	0	0
		This Work[p]	0	0	0
$TS_{\{1\}}$: Stochastic Truncation, 1-bit Slack	3PC	Dalskov [10]	0	10ℓ	3
		This Work	3ℓ	$\mathbf{2\ell}$	**1**
		This Work[p]	3ℓ	0	0
	4PC	Fantastic [11]	0	12ℓ	4
		This Work	4ℓ	$\mathbf{4\ell}$	**1**
		This Work[p]	6ℓ	0	0
$TE_{\{1\}}$: Exact Truncation, 1-bit Slack	3PC	Escudero [15]	$2A_{\ell,t}$	$2A_{\ell,t}$	$2A_{\ell}$
		This Work	$A_{\ell,t}$	$\mathbf{A_{\ell,t}}$	$\mathbf{A_{\ell}}$
		This Work[p]	A_t	A_t	0
	4PC	Fantastic [11]	$3A_{\ell,t}$	$3A_{\ell,t}$	$2A_{\ell}$
		This Work	$A_{\ell,t}$	$\mathbf{A_{\ell,t}}$	$\mathbf{A_{\ell}}$
		This Work[p]	A_t	A_t	0
Trunc. prior to Mult.[d]	3PC	Bicoptor 2.0 [51]	2T	2T	T
		This Work	**T**	**T**	T

Costs are measured in ring elements (ℓ), ℓ-bit and t bit extraction circuits ($A_{\ell,t}$), or truncation primitives (T).

All protocols in Trio and Quad can be converted into online-only protocols with the same number of rounds using the interleaved processing model proposed by [17, 18].

[c] Constant-round online communication [17].

[p] Optimized construction when fusing truncation with certain PPML layers such as ReLU or BatchNorm.

[d] Tweak to truncate two shares prior to multiplication to reduce the probability of truncation failure.

average pooling, public denominators can often be represented with fewer fractional bits while maintaining equivalent precision, thereby reducing the probability of truncation failure.

We also propose a novel *mixed truncation* approach, denoted as $TS_{\{Mix\}}$ (cf. §5), which applies different truncation strategies to different layers of a neural network. Our mixed-truncation approach incurs no communication overhead for truncation in the online phase and matches the accuracy of plaintext VGG-16 inference on the ImageNet dataset, with over 80% accuracy while using shares with a bit length of only 32 bits. In contrast, SOTA methods typically report accuracy only on smaller datasets and require bit lengths of up to 64–80 bits to maintain high accuracy [11].

2 Preliminaries

This section outlines the notations, sharing semantics, threat model, and functionalities used in this work. Since our truncation schemes are built upon the 3PC protocol Trio and the 4PC protocol Quad from [17], we adopt their sharing semantics to ensure consistency and improve readability.

Furthermore, as these protocols follow the function-dependent preprocessing paradigm [37] and use sharing semantics similar to ASTRA [8] and Tetrad [27], our truncation primitives can likely be adapted to these protocols with minimal modifications.

Notations. We denote the set of all parties as $\mathcal{P}$ and refer to the ith party as P_i. A subset of parties, represented by $\mathcal{P}_\Phi$, consists of all parties in the set Φ. For example, $\mathcal{P}_\Phi$, or simply $\mathcal{P}_{i,j}$, represents the subset $\Phi = \{P_i, P_j\}$. Similarly, a value held by all parties in Φ is denoted as x_Φ, or simply $x_{i,j}$ when $\Phi = \{P_i, P_j\}$.

Truncation of a value x by t-bits is denoted as $x^t = \lfloor \frac{x}{2^t} \rfloor$. We define exact truncation as $(x)^t$ and stochastic truncation as $(x)^{st}$. An exact or stochastic truncation scheme that requires a slack of s is denoted as $TE_{\{s\}}$ and $TS_{\{s\}}$, respectively. Here, $s \in \{0, 1, L\}$, where L represents a large, unspecified slack.

Sharing Schemes. We use three different sharing schemes in this work, detailed below. Each scheme operates over an ℓ-bit ring $\mathbb{Z}_{2^\ell}$.

(1) $[\cdot]$-sharing: A value $x \in \mathbb{Z}_{2^\ell}$ is $[\cdot]$-shared among $\mathcal{P}_\Phi$, if each $P_i \in \mathcal{P}_\Phi$ holds x^i such that $\sum_i x^i = x$.
(2) $[\![\cdot]\!]$-sharing: A value $x \in \mathbb{Z}_{2^\ell}$ is $[\![\cdot]\!]$-shared among $\mathcal{P}_\Phi$, if parties in $\mathcal{P}_\Phi$ hold m_x and $[\lambda_x]$ such that $\mathsf{m}_x = x + \lambda_x$.
(3) $\langle\cdot\rangle$-sharing: $\langle x \rangle$ denotes a generic secret sharing of $x \in \mathbb{Z}_{2^\ell}$ without specifying its sharing semantics.

While primitives based on $[\![\cdot]\!]$-sharing are specifically designed for the Trio and Quad protocols in [17], the constructions using $\langle\cdot\rangle$-sharing are more general and can be implemented with any linear secret sharing scheme.

Additionally, $\langle\cdot\rangle^B$ represents Boolean sharing, where addition and multiplication are replaced by XOR and AND gates, respectively, while $\langle\cdot\rangle^A$ denotes arithmetic sharing. The superscript is omitted when the sharing type is clear from context.

Table 2: Sharing semantics for 3PC and 4PC protocols.

	Party	Trio (3PC)	Quad (4PC)
Sharing Semantics $[\![x]\!]$	P_0	λ_x^1, λ_x^2	$\mathsf{m}_x^*, \lambda_x^1, \lambda_x^2$
	P_1	$\mathsf{m}_{x,2}, \lambda_x^1$	$\mathsf{m}_x, \lambda_x^*, \lambda_x^1$
	P_2	$\mathsf{m}_{x,1}, \lambda_x^2$	$\mathsf{m}_x, \lambda_x^*, \lambda_x^2$
	P_3	-	$\lambda_x^*, \lambda_x^1, \lambda_x^2$
	Correlation	$\mathsf{m}_{x,1} = x + \lambda_x^1$	$\lambda_x = \lambda_x^1 + \lambda_x^2$
		$\mathsf{m}_{x,2} = x + \lambda_x^2$	$\mathsf{m}_x = x + \lambda_x$
			$\mathsf{m}_x^* = x + \lambda_x^*$

Table 2 summarizes the sharing semantics for Trio and Quad protocols from [17], which use function-dependent preprocessing [3, 8, 30]. While input-independent λ_x shares are generated non-interactively during preprocessing, computing input-dependent m_x shares may require interaction between parties. Some shares in Quad are solely for verification and are needed only at the protocol's end. For example, communication involving m_x^* is constant-round and does not impact the online phase's round complexity. For further details, we refer readers to [17].

Threat Model. We adopt the threat model of Trio and Quad [17], which assumes an honest-majority setting with at most one corrupted party. Trio's 3PC protocol ensures semi-honest security, while Quad's 4PC protocol provides security with fairness, tolerating malicious corruption. For private inference we assume the client-server model [12] where model- and data owner secretly share their inputs with the 3-4 parties carrying out the computation.

Functionalities. Our constructions utilize cryptographically secure implementations of the Shared Random Value Generator functionality ($\mathcal{F}_{\mathsf{SRNG}}$), enabling a subset of parties $\mathcal{P}_\Phi$ to generate fresh random values without interaction using pseudorandom functions (PRFs). The protocol assumes an initial shared-key setup ($\mathcal{F}_{\mathsf{setup}}$), a standard assumption in most existing protocols [2, 9, 11].

To achieve malicious security in Quad, each party must verify the correctness of received messages. For this, parties utilize a Compare-View functionality, similar to the joint-message passing in SWIFT [25] and the jsnd primitive in Tetrad [27]. We refer readers to [17] for the formal descriptions of Compare-View (Π_{CV}) and sampling shared random values (Π_{SRNG}).

3 Overview of Truncation Approaches

In this section, we provide an overview of the different state-of-the-art truncation approaches that we investigate in this work.

3.1 Stochastic Truncation

Stochastic truncation is the most widely used approach in the MPC literature. It was introduced by SecureML [34] and later adopted by ABY3 [33], and has since become a standard technique in many state-of-the-art MPC frameworks [17, 23, 45, 49].

Stochastic truncation primitives based on these works assume that $\mathcal{P}$ can generate an additive sharing $[x] = x^1 + x^2$ from their existing sharing $\langle x\rangle$, where one subset of parties holds x^1 and another subset holds x^2. Each subset then locally computes $\left(x^1\right)^t$ and $\left(x^2\right)^t$, respectively. Finally, the parties secret-share and add the truncated values to obtain $\langle (x)^{st}\rangle = \langle \left(x^1\right)^t\rangle + \langle \left(x^2\right)^t\rangle$. Figure 1 illustrates this general procedure for stochastic truncation, denoted as $TS_{\{L\}}$ in this work.

Protocol $\Pi_{\mathsf{TS}_{\{L\}}}(\langle x\rangle) \rightarrow \langle (x)^{st}\rangle$

```
(1) Create a $\binom{2}{2}$ $[\cdot]$-sharing of $\langle x\rangle$ denoted by $[x] = x^1 + x^2$.
(2) Compute $\left(x^1\right)^t$ and $\left(x^2\right)^t$ using local truncation and create a
    $\langle\cdot\rangle$-sharing of the two values.
(3) Output $\langle \left(x^1\right)^t\rangle + \langle \left(x^2\right)^t\rangle$.
```

Figure 1: $TS_{\{L\}}$: Stochastic Truncation requiring a large slack [34].

$TS_{\{L\}}$ introduces two types of errors: a small one-off error (e_0) and, with some probability, a larger error (e_1) that leads to truncation failure. The one-off error e_0 causes the truncated value to be either one bit larger or smaller than the corresponding truncated plaintext value. In contrast, the large error e_1 causes the truncated value to differ significantly from the expected value. To illustrate the impact of e_1, we refer to an example from [51]:

$$\begin{aligned}
[\![x]\!] &= 0100\ 1011, \quad \ell = 8, \quad t = 4\\
\lambda_x &= 1110\ 0000,\\
\mathsf{m}_x &= (x + \lambda_x) \bmod 2^8 = 0010\ 1011,\\
\left([\![x]\!]\right)^{st} &= \left((\mathsf{m}_x)^t \bmod 2^8 - \left(\lambda_x\right)^t \bmod 2^8\right) \bmod 2^8\\
&= (0000\ 0010 - 0000\ 1110) \bmod 2^8 = 1111\ 0100
\end{aligned}$$

The actual result of probabilistic truncation in this example is 1111 0100, whereas truncating the plaintext value x yields 0000 0100. This error can significantly impact the accuracy of ML applications. The e_1 error is sometimes referred to as a wrap-around error because a carry bit is falsely propagated through the truncated values, leading to a large error. The closer the actual value x is to the ring modulus 2^ℓ, the higher the probability that $TS_{\{L\}}$ truncation results in truncation failure due to an incorrectly propagated carry bit. More precisely, assuming a two's complement representation and $x \in [0, 2^{\ell_x}) \cup (2^\ell - 2^{\ell_x}, 2^\ell)$ in $\mathbb{Z}_{2^\ell}$, the probability of truncation failure, as analyzed by [51], is given by:

$$P = \frac{1}{2^{\ell - \ell_x - 1}}$$

To mitigate this issue, state-of-the-art frameworks utilize a *slack* mechanism, which increases the utilized ring size 2^ℓ to reduce the probability of this type of error. As a result, an application may need to use a ring $\mathbb{Z}_{2^{64}}$ even in cases where all inputs fit within the ring $\mathbb{Z}_{2^{32}}$ without overflow.

3.2 Stochastic Truncation with Reduced Slack

This variant of stochastic truncation, denoted as $TS_{\{1\}}$, prevents wrap-around errors in $TS_{\{L\}}$ by introducing additional communication. The probability of truncation failure for these schemes with a slack requirement of s is 1 if $\ell - \ell_x - s < 0$ and 0 otherwise. This implies that truncation fails deterministically only when the slack requirement is not met. This represents a significant improvement over $TS_{\{L\}}$, as even values close to the threshold cannot cause truncation failure.

Stochastic truncation with reduced slack (cf. Figure 2) was first proposed by Dalskov et al. [10]. This approach also introduces an e_0 error but eliminates the e_1 error. However, their truncation scheme guarantees correct results only if the most significant bit of x is 0. Escudero et al. [15] addressed this limitation by adding $2^{\ell-1}$ before truncation and subtracting $2^{\ell-t-1}$ after truncation. This trick ensures that the most significant bit of x is 0 but introduces a slack of 1 bit to maintain correctness.

Protocol $\Pi_{\mathsf{TS}_{\{1\}}}(\langle x\rangle) \rightarrow \langle (x)^{st}\rangle$

```
(1) Add $2^{\ell-1}$ to $\langle x\rangle$ to ensure $\mathsf{MSB}(x) = 0$.
(2) Generate $\ell$ random shared bits $\langle r_i\rangle^B$ and compute
    $\langle r\rangle^A \leftarrow \sum_i \langle r_i\rangle \cdot 2^i$.
(3) Open $c \leftarrow \langle x\rangle + \langle r\rangle$ and compute $c' \leftarrow ((c)^t) \bmod 2^{\ell-t-1}$.
(4) Compute $\langle b\rangle \leftarrow \langle r_{\ell-1}\rangle \oplus \mathsf{MSB}(c)$.
(5) Compute $\langle y\rangle = c' - \sum_{i=t}^{\ell-2} \langle r_i\rangle \cdot 2^{i-t} + \langle b\rangle \cdot 2^{\ell-t-1}$
(6) Output $\langle y\rangle - 2^{\ell-t-1}$.
```

Figure 2: $TS_{\{1\}}$: Stochastic Truncation requiring 1 bit slack [10].

The intuition behind $TS_{\{1\}}$ is that the parties generate a new mask for x using $\langle r\rangle$ and replace the existing mask of x by opening $\langle x\rangle + \langle r\rangle$ in step 2. By computing

$$c' \leftarrow ((c)^t) \bmod 2^{\ell-t-1},$$

the parties truncate the value c to retain the desired t fractional bits while discarding the t most significant bits of c. These discarded bits could be affected by truncation failure if relying on $TS_{\{L\}}$.

Steps 3-4 leverage the fact that the parties also hold a bit decomposition of the mask $\langle r\rangle$, allowing them to deterministically recover the t most significant bits of c'.

3.3 Exact Truncation

Exact Truncation computes $(x)^t$ deterministically, without causing an e_0 and e_1 error. Figure 3 shows a general procedure for exact truncation without requiring any slack, denoted as $TE_{\{0\}}$.

Protocol $\Pi_{\mathsf{TE}_{\{0\}}}(\langle x\rangle) \rightarrow \langle (x)^t\rangle$

(1) Use Π_{A2B} to convert $\langle x\rangle^A$ to $\langle x\rangle^B$.
(2) Compute $\langle x'\rangle = \langle x\rangle^t$ using an arithmetic right shift of $\langle x\rangle^B$ by t bits using only local bit assignments.
(3) Output $\Pi_{\mathsf{B2A}}(\langle x'\rangle^B)$.

Figure 3: $TE_{\{0\}}$**: Exact Truncation without requiring any slack.**

The protocol begins by converting the arithmetic sharing $\langle x\rangle^A$ into a boolean sharing $\langle x\rangle^B$ using arithmetic-to-binary conversion (Π_{A2B}). Since $\langle x\rangle^B$ represents an XOR-sharing of bits $x[0], \ldots, x[\ell-1]$, where the most significant bit (MSB) is $x[0]$, performing an arithmetic right shift by t bits corresponds to locally setting

$$\langle x'\rangle^B[i] = \langle x\rangle^B[i-t] \qquad \text{for } i \geq t,$$
$$\langle x'\rangle^B[i] = \langle x\rangle^B[0] \qquad \text{for } i < t.$$

This operation effectively shifts all bits t positions to the right and sets the vacated bits with the original sign bit. Finally, the parties convert the boolean sharing of $(x)^t$ back into an arithmetic sharing using binary-to-arithmetic conversion (Π_{B2A}). The Π_{A2B} and Π_{B2A} protocols require evaluating a boolean addition circuit, which can be implemented using one of the variants described in §1.1.

3.4 Exact Truncation with Slack

Fantastic Four [11] proposed a more efficient exact truncation scheme based on additive sharing, replacing the need for full-bit adders with bit extraction circuits. These circuits can be implemented using the same number of rounds but often require fewer gates. The resulting approach requires a slack of 1 bit and is referred to as $TE_{\{1\}}$ in this work. For $f(x) = x - (x \bmod 2^\ell)$, truncation of $[x]$ with $\mathsf{MSB}(x) = 0$ is computed as follows:

$$(x)^t = \sum_{i=0}^{n-1} x^i/2^t + \left(\sum_i (x^i \bmod 2^t)\right)/2^t - f\left(\sum_i x^i\right)/2^t$$

The intuition behind this formula is as follows: The first term computes the truncation of each share individually, similar to $TS_{\{L\}}$. The second term corrects the one-off error (e_0), while the third term corrects the wrap-around error (e_1) introduced by the first term. The first term in the sum is computed locally by each party by directly dividing its share. The remaining terms are computed interactively in the boolean domain using binary adders.

The authors also observed that the second term, $\sum_i (x^i \bmod 2^m)/2^m$, is always smaller than n, where n is the number of unique shares held by the parties. Therefore, a bit extraction circuit that computes the carry bits at positions $[t, t+\log n-1]$ suffices to obtain the result. Similarly, the third term contains non-zero bits only in the $\log(n)$ most significant bits, requiring another bit extraction circuit on $\log(n) + \ell$ bits. Since extracting the carry bits of t-bit terms is computationally cheaper than extracting the carry bits of ℓ-bit terms, the last term—responsible for correcting e_1—accounts for most of the amortized communication complexity.

4 Efficient Truncation Protocols

This section details the efficient implementation of the truncation approaches from §3 within the Trio 3PC and Quad 4PC protocols from [17]. Since our constructions replicate the exact functionalities of the respective SOTA truncation approaches, they maintain the same slack requirements, probability of truncation failure, and security properties. However, by leveraging sharing semantics, local preprocessing, and subset-sharing primitives, our constructions significantly reduce communication complexity compared to a naive adaptation of these approaches (cf. Table 1).

4.1 Truncation-Related Primitives

To construct efficient truncation protocols, we observe that many existing approaches benefit from parties holding a $\binom{2}{2}$ additive sharing, where one subset of parties holds x^1 and a disjoint subset holds x^2, with $x = x^1 + x^2$. Using this $[\cdot]$-sharing, parties follow a three-step process: first, they locally apply modulus or truncation operations to x^1 and x^2; second, they share the modified values among $\mathcal{P}$; and third, they aggregate the modified values to obtain the final result.

The Trio and Quad protocols from [17] are particularly well-suited fro this approach, as they naturally support decomposing x into its $[\cdot]$-shares. In these protocols, one subset of parties locally obtains $-\lambda_x$, while a disjoint subset obtains m_x such that $x = (-\lambda_x)+\mathsf{m}_x$. For instance, in Trio, P_0 computes $-\lambda_x = -(\lambda_x^1+\lambda_x^2)$, while both P_1 and P_2 can locally compute m_x as follows: P_1 computes $\mathsf{m}_x = \mathsf{m}_{x,2} + \lambda_x^1$ and P_2 computes $\mathsf{m}_x = \mathsf{m}_{x,1} + \lambda_x^2$. In Quad, each party holds either m_x or λ_x by default. We refer to parties that can locally obtain m_x as $\mathcal{P}_{\mathsf{m}_x}$ and those that can obtain λ_x as $\mathcal{P}_{\lambda_x}$. Note that λ_x is an input-independent share that can be generated non-interactively during the preprocessing phase.

Subset-sharing. To efficiently generate $[\![\cdot]\!]$-sharings of $[\cdot]$-shared values among $\mathcal{P}$, we leverage the concept of subset-sharing proposed by [44]. Subset sharing allows specific subsets of parties to generate secret-shares of a joint value among $\mathcal{P}$. We primarily consider scenarios which require generation of $[\![\cdot]\!]$-shares of a value that is available to some of the parties, either during the preprocessing or online phase.

Figures 4 and 5 present the subset-sharing primitives for 3PC, while Figures 6 and 7 present the 4PC setting. These primitives minimize the communication overhead of secret sharing by exploiting two key optimizations: (1) certain shares can be locally obtained using Π_{SRNG}, and (2) other shares can be directly set to 0.

Protocol $\Pi_{\mathsf{SH}_{3\mathsf{PC}}}(u, P_0) \rightarrow [\![u]\!]$

Preprocessing:

(1) P_0 samples λ_u^1 with P_1 using Π_{SRNG}.
(2) P_0 computes $\lambda_u^2 = -\lambda_u^1 - u$ and sends λ_u^2 to P_2.

Online:

$\mathcal{P}_{1,2}$ set their input-dependent shares to 0.

Figure 4: 3PC Subset-Sharing of value u **held by** $\mathcal{P}_{\lambda_x} = P_0$**.**

Protocol $\Pi_{\mathsf{SH}_{\mathsf{3PC}}}(u, \mathcal{P}_{1,2}) \rightarrow [\![u]\!]$

Preprocessing:

All parties set their input-independent shares to 0.

Online:

$\mathcal{P}_{1,2}$ set their input-dependent shares to u.

Figure 5: 3PC Subset-Sharing of value u held jointly by $\mathcal{P}_{\mathsf{m}_x} = \mathcal{P}_{1,2}$.

The 4PC primitives incorporate the verify-send technique from SWIFT [25], where, among the two parties holding a message, one sends it to the recipient while the other parties verify its correctness using a Compare-View protocol Π_{CV}. The honest-majority assumption ensures that the recipient either receives the correct message or aborts the protocol. Additionally, the Quad protocols leverage that the value m_x^* is only required by P_0 at the end of the protocol. As a result, the related online communication does not increase the round complexity.

Table 3 summarizes the communication complexity of our subset-sharing primitives. The subset of parties that initially holds the value to be shared is indicated in the primitive's signature. Notably, the subset-sharing primitive by $\mathcal{P}_{\lambda_x}$ is applicable only to values available in the preprocessing phase.

Protocol $\Pi_{\mathsf{SH}_{\mathsf{4PC}}}(u, \mathcal{P}_{0,3}) \rightarrow [\![u]\!]$

Preprocessing:

(1) $\mathcal{P}_{0,3}$ samples λ_u^1 with P_1 using Π_{SRNG}.
(2) $\mathcal{P}_{0,3}$ computes $\lambda_u^2 = -\lambda_u^1 - u$ and verify-sends λ_u^2 to P_2.
(3) $\mathcal{P}_{1,2,3}$ set λ_u^* to 0.

Online:

P_0 sets m_u^* to x while $P_{1,2}$ set m_{uS} to 0.

Figure 6: 4PC Subset-Sharing of value u jointly held by $\mathcal{P}_{\lambda_x} = \mathcal{P}_{0,3}$.

Protocol $\Pi_{\mathsf{SH}_{\mathsf{4PC}}}(u, \mathcal{P}_{1,2}) \rightarrow [\![u]\!]$

Preprocessing:

(1) $\mathcal{P}_{1,2,3}$ sample λ_u^* using Π_{SRNG}.
(2) The parties set all remaining input-independent shares to 0.

Online:

(1) $\mathcal{P}_{1,2}$ set their input-dependent share $\mathsf{m}_u = u$.
(2) $\mathcal{P}_{1,2}$ verify-send $\mathsf{m}_x^* = u + \lambda_u^*$ to P_0 as part of constant-round communication.

Figure 7: 4PC Subset-Sharing of value u jointly held by $\mathcal{P}_{\mathsf{m}_x} = \mathcal{P}_{1,2}$.

Table 3: Communication complexity of subset-sharing primitives.

Setting	Subset holding u	PRE	ON	Rounds
3PC	$\mathcal{P}_{\lambda_x} = P_0$	ℓ	0	0
	$\mathcal{P}_{\mathsf{m}_x} = \mathcal{P}_{1,2}$	0	0	0
4PC	$\mathcal{P}_{\lambda_x} = P_{0,3}$	ℓ	0	0
	$\mathcal{P}_{\mathsf{m}_x} = \mathcal{P}_{1,2}$	0	ℓ	0

4.2 Truncation Approaches in Trio and Quad

This section provides formal details of our constructions of the truncation approaches from §3 within the Trio and Quad protocols. Since our work does not improve upon a naive adaptation of exact truncation $TE_{\{0\}}$ in Trio and Quad, we omit its formal details from this section. Additionally, the protocols are presented in a generic manner, allowing them to be instantiated using either Trio or Quad.

4.2.1 $TS_{\{L\}}$: Stochastic Truncation. Figure 8 presents our construction of $TS_{\{L\}}$ in Trio and Quad. While our construction does not offer significant improvements over the SOTA (cf. Table 1), we include it for completeness, as stand-alone truncation primitives have not yet been proposed for Trio and Quad.

Protocol $\Pi_{\mathsf{TS}_{\{\mathsf{L}\}}}([\![x]\!] \rightarrow [\![(x)^{st}]\!])$

(1) $\mathcal{P}_{\lambda_x}$ computes $(x^1)^t = (-\lambda_x)^t$ and executes $\Pi_{\mathsf{SH}}(\mathcal{P}_{\lambda_x}, (x^1)^t)$.
(2) $\mathcal{P}_{\mathsf{m}_x}$ computes $(x^2)^t = (\mathsf{m}_x)^t$ and executes $\Pi_{\mathsf{SH}}(\mathcal{P}_{\mathsf{m}_x}, (x^2)^t)$.
(3) Output $[\![(x^1)^t]\!] + [\![(x^2)^t]\!]$.

Figure 8: $TS_{\{L\}}$ in Trio and Quad.

In Trio, parties can locally generate an additive secret sharing $[x]$ using their existing shares. Specifically, $\mathcal{P}_{1,2}$ can compute $\mathsf{m}_x = \mathsf{m}_{x,1} + \lambda_x^2 = \mathsf{m}_{x,2} + \lambda_x^1$, while P_0 computes $-\lambda_x = -(\lambda_x^1 + \lambda_x^2)$. Similarly, in Quad, an additive sharing is naturally formed, where $\mathcal{P}_{1,2}$ hold $\mathsf{m}_x = x + \lambda_x$, and $\mathcal{P}_{0,3}$ hold λ_x.

Leveraging this insight, the parties can locally truncate their respective shares of x and distribute the truncated values using the subset-sharing primitives.

4.2.2 $TS_{\{1\}}$: Stochastic Truncation with Reduced Slack. Dalskov et al. [10] and Fantastic Four [11] provide tailor-made constructions of $TS_{\{1\}}$ for the 3PC and 4PC settings, respectively. Naively reimplementing these primitives in Trio and Quad results in similar communication complexity due to the need to generate and open shared random bits, as shown in steps 2-3 of Figure 2. However, we show how to construct a slack-free truncation protocol in Trio and Quad with up to four times lower round complexity and up to five times lower online complexity (cf. Table 1) by replacing these communication-intensive operations with local computations and leveraging our subset-sharing primitives.

Protocol $\Pi_{\mathsf{TS}-1}([\![x]\!]) \rightarrow ([\![x]\!])^{st}$

(1) Add $2^{\ell-1}$ to $[\![x]\!]$ to ensure $\mathsf{MSB}(x) = 0$.
(2) $\mathcal{P}_{\lambda_x}$ compute $r' = \sum_{i=t}^{\ell-2} \lambda_{x,i} \cdot 2^{i-t}$ and $r_{\ell-1} = \mathsf{MSB}(\lambda_x)$.
(3) $\mathcal{P}_{\mathsf{m}_x}$ compute $c' = (\mathsf{m}_x)^t \bmod 2^{\ell-t-1}$ and $\mathsf{MSB}(c) = \mathsf{MSB}(\mathsf{m}_x)$.
(4) Generate $[\![\cdot]\!]^A$-sharings $[\![r']\!], [\![r_{\ell-1}]\!], [\![c']\!], [\![MSB(c)]\!]$ using Π_{SH}.
(5) Compute $[\![b]\!]^A = [\![r_{\ell-1}]\!]^A \oplus [\![MSB(c)]\!]^A$.
(6) Compute $[\![y]\!] = [\![c']\!] - [\![r']\!] + [\![b]\!] \cdot 2^{\ell-t-1}$.
(7) Output $[\![y]\!] - 2^{\ell-t-1}$.

Figure 9: $TS_{\{1\}}$ in Trio and Quad.

Figure 9 presents our construction of $TS_{\{1\}}$. To implement $TS_{\{1\}}$, we exploit the fact that $\mathcal{P}_{\mathsf{m}_x}$ can locally define $c = \mathsf{m}_x$, while $\mathcal{P}_{\lambda_x}$ can locally define $r = \lambda_x$. This observation allows parties to bypass all communication-related operations in steps 1 and 2 of Figure 2, including generating doubly authenticated bits $[\![r]\!]$ and even opening the value c. Additionally, parties can precompute certain operations locally, such as $r' = \sum_{i=t}^{\ell-2} \lambda_i \cdot 2^{i-t}$ and $c' = (\mathsf{m}_x)^t \bmod 2^{\ell-t-1}$, to avoid computing these expressions jointly. Finally, parties use our efficient subset-sharing primitives to share the locally modified shares and compute the final result, analogous to the original protocol. The XOR operation in step 4 can be evaluated in the arithmetic domain using the identity $a \oplus b = a + b - 2ab$.

4.2.3 $TE_{\{1\}}$: *Exact Truncation with Slack.*

The main drawback of Fantastic Four's [11] exact truncation primitive $TE_{\{1\}}$ is that the protocol requires four shares per party. As a result, the bit extraction circuits must be evaluated using a tree-based approach, involving three adders and two levels of addition.

In contrast, using a $\binom{2}{2}$ additive sharing would reduce the number of bits to extract from four to two while also requiring only a single adder per term. Note that constructing a $\binom{2}{2}$ additive sharing in Trio and Quad is straightforward, as parties can efficiently create sharings of m_x and $-\lambda_x$ with minimal communication overhead using subset-sharing techniques.

Figure 10 illustrates the optimized $TE_{\{1\}}$ protocol, leveraging the sharing semantics of Trio and Quad. Steps 2–4 establish the necessary sharings to compute all sums in the truncation formula. Steps 5–6 compute the potential non-zero bits in the Boolean domain. Step 7 then converts the carry bits into the arithmetic domain, and step 8 aggregates the terms to derive the final result.

Note that combining the one-off error correction from Figure 10 with the more efficient wrap-around error correction from $TS_{\{1\}}$ (cf. Figure 9) could further reduce communication complexity. We leave the exploration of this combined approach for future work. Finally, we implement the remaining truncation scheme $TE_{\{0\}}$, introduced in §3, using a straightforward approach based on Figure 3.

Protocol $\Pi_{\mathsf{TE}_{\{1\}}}(\llbracket x \rrbracket) \rightarrow \llbracket (x)^t \rrbracket$

(1) Add $2^{\ell-1}$ to $\llbracket x \rrbracket$ to ensure $\mathsf{MSB}(x) = 0$.
(2) $\mathcal{P}_{\mathsf{m}_x}$ generate $\llbracket \mathsf{m}_x/2^t \rrbracket$, while $\mathcal{P}_{\lambda_x}$ generate $\llbracket -\lambda_x/2^t \rrbracket$ using Π_{SH}. Parties compute $\llbracket x/2^t \rrbracket = \llbracket \mathsf{m}_x/2^t \rrbracket + \llbracket -\lambda_x/2^t \rrbracket$.
(3) $\mathcal{P}_{\mathsf{m}_x}$ generate $\llbracket \mathsf{m}_x \bmod 2^t \rrbracket^B$, while $\mathcal{P}_{\lambda_x}$ generate $\llbracket -\lambda_x \bmod 2^t \rrbracket^B$ using Π_{SH}.
(4) $\mathcal{P}_{\mathsf{m}_x}$ generate $\llbracket \mathsf{m}_x \rrbracket^B$, while $\mathcal{P}_{\lambda_x}$ generate $\llbracket -\lambda_x \rrbracket^B$ using Π_{SH}.
(5) Calculate the carry bit $\llbracket b_1 \rrbracket^B$ for bit position $t+1$ of $\llbracket \mathsf{m}_x \bmod 2^t \rrbracket^B + \llbracket -\lambda_x \bmod 2^t \rrbracket^B$ using a t-bit carry adder.
(6) Calculate the carry bit for bit position $\ell + 1$ of $\llbracket b_2 \rrbracket^B$ of $\llbracket \mathsf{m}_x \rrbracket^B + \llbracket -\lambda_x \rrbracket^B$ using an ℓ-bit carry adder.
(7) Convert $\llbracket b_1 \rrbracket^B$ and $\llbracket b_2 \rrbracket^B$ to $\llbracket b_1 \rrbracket^A$ and $\llbracket b_2 \rrbracket^A$ using Π_{Bit2A}.
(8) Compute $\llbracket y \rrbracket = \llbracket x/2^t \rrbracket + \llbracket b_1 \rrbracket^A - \llbracket b_2 \rrbracket^A \cdot 2^{\ell-t}$.
(9) Output $\llbracket y \rrbracket - 2^{\ell-t-1}$.

Figure 10: $TE_{\{1\}}$ in Trio and Quad.

5 Applying Truncation in PPML

In this section, we propose how to efficiently integrate the different truncation approaches into the PPML inference of neural network layers. We observe that the properties of several layers allow us to reduce the slack size required by a truncation scheme as well as reduce its communication complexity.

Linear Layers and Batch Normalization. Linear layers such as fully connected layers and convolutional layers require matrix multiplication of fixed-point shares. Thus, each output share needs to be truncated. Batch Normalization computes $y(x) = \frac{x-\mu}{\sqrt{\sigma^2+\epsilon}} \cdot \gamma + \beta$ where the parameters $\mu, \sigma, \gamma, \beta$ are model parameters obtained during training, and ϵ is a small public constant to avoid division by zero. Thus, during inference, the party holding the model parameters locally computes $\hat{\sigma} = \gamma \cdot \frac{1}{\sqrt{\sigma^2+\epsilon}}$ and shares it along with μ and β among the parties. Using these shares, the parties can compute the layer with a single fixed-point multiplication. When using $TS_{\{L\}}$, we exploit that truncation can be integrated into the multiplication protocols of Trio and Quad at no additional communication costs [17]. The formal protocol is described in the authors' work.

As Batch Normalization typically appears directly after a linear layer, $TS_{\{1\}}$ of the linear layer can be fused with the multiplication in Batch Normalization using multi-input multiplication gates [37] to reduce the $TS_{\{1\}}$ overhead in round complexity to 0. These can be further optimized to multi-input scalar products [4] to also reduce the overhead in online communication to 0. To do so, observe that step 3 in $\Pi_{\mathsf{TS-1}}$ requires an XOR operation of the two shares $\llbracket m \rrbracket^A = \llbracket r_{l-1} \rrbracket$ and $\llbracket n \rrbracket^A = \llbracket \mathsf{MSB}(c) \rrbracket$ followed by a multiplication with public value $k = 2^{\ell-t-1}$. The results need to be added to $\llbracket o \rrbracket = \llbracket c' \rrbracket - \llbracket r' \rrbracket - 2^{\ell-t-1} - \llbracket \mu \rrbracket$ to obtain the first factor to compute the batch normalization. Hence, the parties wish to compute the following expression to obtain the layer output y of batch normalization which can be expressed as a single scalar product consisting of one two-input multiplication and one three-input multiplication in a single round of communication as follows:

$$\begin{aligned}\llbracket y \rrbracket &= \llbracket \hat{\sigma} \rrbracket \cdot (\llbracket o \rrbracket + (\llbracket m \rrbracket \oplus \llbracket n \rrbracket)k + \beta \\ &= \llbracket \hat{\sigma} \rrbracket \cdot \llbracket o \rrbracket + \llbracket \hat{\sigma} \rrbracket \cdot (\llbracket km \rrbracket + \llbracket kn \rrbracket - 2\llbracket km \rrbracket \cdot \llbracket kn \rrbracket) + \beta \\ &= \llbracket \hat{\sigma} \rrbracket (\llbracket o \rrbracket + \llbracket km \rrbracket + \llbracket kn \rrbracket) + \llbracket -2\hat{\sigma} \rrbracket \cdot \llbracket km \rrbracket \cdot \llbracket kn \rrbracket + \beta\end{aligned}$$

Activation Functions. Normalization layers or linear layers are typically followed by an activation function. The most frequently used activation function in convolutional neural networks is ReLU. The ReLU operation is defined as $\mathsf{ReLU}(x) = \max(x, 0)$. To perform a ReLU operation, parties convert $\langle x \rangle^A$ to $\langle x \rangle^B$, evaluate a sign bit extraction circuit, and negate the result to obtain $\mathsf{DReLU}(x) = \langle \neg \mathsf{MSB}(x) \rangle^B$. ReLU(x) can then be computed as $\langle \mathsf{DReLU(x)} \rangle^B \cdot \langle x \rangle^A$ using Bit Injection [33].

Protocol $\Pi_{\mathsf{ReLU+TE}_{\{0\}}}(\langle x \rangle) \rightarrow \langle \mathsf{ReLU}((x)^t) \rangle$

(1) Execute steps 2-5 of $\Pi_{\mathsf{TE}_{\{1\}}}$.
(2) Execute step 6 of $\Pi_{\mathsf{TE}_{\{1\}}}$ but also extract $\llbracket \mathsf{MSB}(x) \rrbracket^B$ as part of the same circuit.
(3) Execute step 7-8 of $\Pi_{\mathsf{TE}_{\{1\}}}$ and multiply the result with $\llbracket \mathsf{MSB}(x) \rrbracket^B$ using Π_{BitInj}.

Figure 11: ReLU with exact truncation ($TE_{\{0\}}$).

All truncation schemes can benefit from delaying the truncation of the layer prior to the ReLU operation, both in terms of reduced slack and communication overhead. The slack-related benefit from delaying truncation until the next ReLU layer is that after an activation all negative values are guaranteed to be 0. As truncating 0 has a negligible probability of truncation failure, this optimization significantly reduces the number of truncation failures in PPML where negative values are as common as positive values. Additionally, the $TE_{\{1\}}$ and $TS_{\{1\}}$ schemes can benefit from the ReLU operation as they do not need to respect their non-negativity constraint: In case of truncation failure, the ReLU operations set the output share to 0 provided that ReLU is calculated based on the untruncated share.

Consequently, the parties can omit the addition and subtraction operations required by the $TE_{\{1\}}$ and $TS_{\{1\}}$ schemes which transform them into $TE_{\{0\}}$ and $TS_{\{0\}}$ schemes, respectively.

The performance-related benefit of delaying truncation applies to the $TS_{\{1\}}$, $TE_{\{1\}}$, and $TE_{\{0\}}$ schemes. Trivially, ReLU can be fused with little communication overhead with $TE_{\{0\}}$ as proposed by [22] by performing a full bit decomposition, applying truncation and ReLU in the boolean domain, and performing a full bit composition to obtain the result.

However, we observe that ReLU can also be merged with the more efficient $TE_{\{1\}}$ and $TS_{\{1\}}$ schemes. The conversion and bit extraction of ReLU can be fused without additional overhead into $\Pi_{\mathsf{TE}_{\{1\}}}$ by letting the carry adder in step 6 of the protocol (cf. Figure 10) also compute the sign bit of $\llbracket x \rrbracket^B$. Hence, the only communication overhead of adding a ReLU operation to the truncation primitive is performing a bit injection which is typically similarly complex to performing a single multiplication in $\mathbb{Z}_{2^\ell}$. Figure 11 describes the protocol for fusing ReLU with $\Pi_{\mathsf{TE}_{\{1\}}}$. Note that $\Pi_{\mathsf{TE}_{\{1\}}}$ can also implement $TS_{\{1\}}$ by skipping all computations related to computing $\llbracket b_1 \rrbracket$ which includes steps 3,5, and one Π_{Bit2A} operation (cf. Figure 10). When optimizing for communication rounds, a generic way of fusing ReLU with $\Pi_{\mathsf{TS}_{\{1\}}}$ is to compute DReLU on the untruncated share $\langle x \rangle$ while computing $\langle (x)^t \rangle$ in parallel to the ReLU computation. The parties then bit inject the result of $\mathsf{DReLU}(\langle x \rangle)$ into the truncated share $\langle (x)^t \rangle$ to obtain $\mathsf{ReLU}(\langle (x)^t \rangle)$ without any overhead in round complexity.

We can design a more efficient protocol for Trio and Quad by fusing the bit injection performed during ReLU with the online communication required by $\Pi_{\mathsf{TS}_{\{1\}}}$ similar to our approach during batch normalization. An additional challenge here is that the output of ReLU prior to Bit Injection is given in the boolean domain and thus the process is more involved than simply utilizing multi-input scalar products. We observe that the online phases of bit injection and $\Pi_{\mathsf{TS}_{\{1\}}}$ can be merged at no additional communication cost or round complexity.

Fusing Truncation and Bit Injection. Similar to our fused Batch Normalization approach we define $\llbracket m \rrbracket^A = \llbracket r_{l-1} \rrbracket$ and $\llbracket n \rrbracket^A = \llbracket \mathsf{MSB}(c) \rrbracket$, The shares need to be first XOR-ed, then multiplied with $k = 2^\ell - t - 1$, and finally added to $\llbracket o \rrbracket = \llbracket c' \rrbracket - \llbracket r' \rrbracket - 2^{\ell-t-1}$ to obtain the first factor to compute the fused bit injection. The second factor $\llbracket x \rrbracket^B$ is the negated most-significant bit of the untruncated $\llbracket x \rrbracket$ obtained during the ReLU operation. The output $\llbracket y \rrbracket$ of fusing truncation and bit injection is thus given by.

$$
\begin{aligned}
\llbracket y \rrbracket &= \llbracket x \rrbracket^B \cdot (\llbracket o \rrbracket^A + (\llbracket m \rrbracket^A \oplus \llbracket n \rrbracket^A)k) \\
&= \llbracket x \rrbracket^B \cdot (\llbracket o \rrbracket + \llbracket km \rrbracket + \llbracket kn \rrbracket - 2\llbracket km \rrbracket \cdot \llbracket kn \rrbracket) \\
&= (\mathsf{m}_x + \lambda_x - 2\mathsf{m}_x\lambda_x) \cdot (\llbracket o \rrbracket + \llbracket km \rrbracket + \llbracket kn \rrbracket - 2\llbracket km \rrbracket\llbracket kn \rrbracket) \\
&= (\mathsf{m}_x + \lambda_x - 2\mathsf{m}_x\lambda_x)(\mathsf{m}_o - \lambda_o + \mathsf{m}_{km} - \lambda_{km} + \mathsf{m}_{kn} - \lambda_{kn} \\
&\quad - 2((\mathsf{m}_{km} - \lambda_{km})(\mathsf{m}_{kn} - \lambda_{kn})) \\
&= (\mathsf{m}_x + \lambda_x - 2\mathsf{m}_x\lambda_x)(\mathsf{m}_o - \lambda_o + \mathsf{m}_{km} - \lambda_{km} + \mathsf{m}_{kn} - \lambda_{kn} \\
&\quad - 2(\mathsf{m}_{km}\mathsf{m}_{kn} - \mathsf{m}_{km}\lambda_{kn} - \mathsf{m}_{kn}\lambda_{km} + \lambda_{km}\lambda_{kn}))
\end{aligned}
$$

When fully expanding the last equation the parties obtain different combinations of input-dependent and input-independent subterms, e.g., $\mathsf{m}_x \cdot \mathsf{m}_o$, $\lambda_x \cdot \lambda_o$, $\mathsf{m}_x \cdot \lambda_o$. Note that all input-dependent terms can be computed locally by $\mathcal{P}_{\mathsf{m}_x}$. To also evaluate the input-independent terms, they need to obtain additive shares of all relevant λ-terms from $\mathcal{P}_{\lambda_x}$ in the preprocessing phase, use these to calculate an additive sharing $[y]$ and reshare the result to all parties to obtain $\llbracket y \rrbracket$. The complete procedure is described in §A.

Pooling Layers. Out of the pooling layers, only average pooling requires truncation. Computing an average in MPC requires a division operation, which is not natively supported in ring-based MPC. However, since the divisor d is public, we can approximate the division by multiplying the input $\langle x \rangle$ with the FPA representation of the reciprocal $r = 1/d$. We can exploit several slack-related optimizations to reduce the probability of truncation failure of that multiplication in average pooling, mainly by exploiting that d is a public value.

Following our proposed approach of computing an average naively would require to approximate r using t bits of precision followed by computing $\langle y \rangle = r \cdot \langle x \rangle$ with $2t$ fractional bits which leads to the same probability of truncation failure as the multiplication of two secret shares. However, we observe that common denominators in average pooling are powers of two with the most common denominator being 4 resulting from a kernel size of 2x2. For $d = 2^k$, the reciprocals can be expressed with k fractional bits without any loss of precision. For denominators that are not powers, we can exploit that the denominator in FPA is approximated using t fractional bits but not all of these bits are significant. For instance, $r = 1/9$ resulting from a kernel size of 3x3 is approximated as 00111000 for $t = 8$ but can be expressed as 111000 for $t = 6$ without any loss of precision. Finally, parties can also exploit that when a reciprocal is between two FPA approximations, choosing the one with lower precision reduces precision by less than 2^{-t} but still reduces the probability of truncation failure by a factor of 2. Hence, parties may additionally choose a threshold to decide when to use an approximation with fewer fractional bits that introduce loss of precision. Note that several networks such as VGG-16 or ResNet architectures use adaptive average pooling which dynamically determines the kernel size of the average pooling layer. These layers frequently create kernels of size 1x1 which results in no required truncation. Figure 12 describes the protocol for computing a division with a reduced probability of truncation failure that includes all described considerations.

Protocol $\Pi_{\mathsf{Division}}(\langle x \rangle, d, t, \mathsf{threshold}) \rightarrow \langle y \rangle \approx x/d$

(1) Approximate $r_t \approx 1/d$ and $r_{t-1} \approx 1/d$ using t and $t-1$ fractional bits in FPA respectively.
(2) Calculate (using floating-points) $e_t = |1/d - r_t|$ and $e_{t-1} = |1/d - r_{t-1}|$.
(3) While $e^t - e^{t-1} \leq \mathsf{threshold}$, decrement t. If $t = 0$, output $\langle x \rangle$.
(4) Compute $\langle y \rangle = r_t \cdot \langle x \rangle$.
(5) Output $(\langle y \rangle)^t$.

Figure 12: Division with reduced probability of truncation failure.

Finally, note that average pooling typically follows after a ReLU layer. Hence, $TE_{\{1\}}$ and $TS_{\{1\}}$ can be implemented without the 1-bit slack requirement thus leading to $TE_{\{0\}}$ and $TS_{\{0\}}$ schemes, respectively. Given that we also achieved 0 bits of slack by delaying the truncation of the layer prior to ReLU, all common neural network architectures that do not use Batch Normalization such as AlexNet, LeNet5, and VGG-16 are completely evaluated without any slack using $TE_{\{1\}}$ and $TS_{\{1\}}$ schemes while layers that use Batch Normalization such as ResNet architectures achieve 0 bits of slack in more than half of all layers.

Mixed Truncation. Given the introduced optimizations, we propose a mixed truncation strategy that combines the benefits of all truncation schemes depending on the PPML layer type. We observed that average pooling deals with small public reciprocals $r \leq 1/4$ that can additionally often be expressed with a smaller fixed-point multiplicator without causing any loss of precision. Hence, this multiplication is an optimal candidate for $TS_{\{L\}}$. In neural network architectures, pooling is typically followed by a linear layer, in some cases followed by a normalization layer, and finally by a ReLU layer. If the linear layer is followed by a Batch Normalization layer, we can apply our proposed $TS_{\{1\}}$ optimization to merge the truncation of the linear layer with the multiplication in Batch Normalization without any overhead in round complexity or online complexity. The untruncated output of the layer prior to ReLU is then truncated within the Bit injection operation required by ReLU as described previously. Observe that the total overhead of the mixed truncation strategy to the forward pass is 0 in terms of round complexity and online communication complexity. Table 4 shows which truncation approach to apply in each layer to the input $[\![x]\!]$ and output $[\![y]\!]$ of the layer.

Table 4: Mixed truncation strategy.

Layer	trunc($[\![x]\!]$)	trunc($[\![y]\!]$)	Optimization
MaxPool	-	-	-
AvgPool	-	$TS_{\{L\}}$	Reduced t[a]
Linear	-	Delay	-
BN	$TS_{\{1\}}$	Delay	Fuse $TS_{\{1\}}$ & BN
ReLU	$TS_{\{1\}}$	-	Fuse ReLU & $TS_{\{1\}}$ [a]

[a] Including slack-based optimization.

Figure 13 shows how our mixed truncation strategy is applied to a a typical sequence of layers—pooling, convolution, batch normalization, and ReLU—as commonly used in CNN architectures like ResNets. Importantly, with our proposed protocols, slack-related optimizations do not introduce any communication costs, while fused operations introduce no additional overhead during the online phase. The figure also shows the number of resulting fractional bits in the input matrix X after each layer in the forward pass. Our truncation strategy ensures that whenever a matrix ends up with twice the desired number of fractional bits (denoted X^{2f}), it is truncated before being used in any subsequent operation involving the multiplication of two fixed-point numbers.

6 Systematic Evaluation

In this section, we show the results of our large-scale comparison of the different truncation approaches. Our evaluation is based on our implementation of all truncation schemes in the open-source HPMPC framework [16] while we utilize PyTorch for plaintext training and inference. In §6.1, we evaluate the impact of our truncation optimizations tailor-made for PPML, as discussed in §5. This is followed by §6.2, where all the truncation approaches considered in this work are evaluated across various models and datasets.

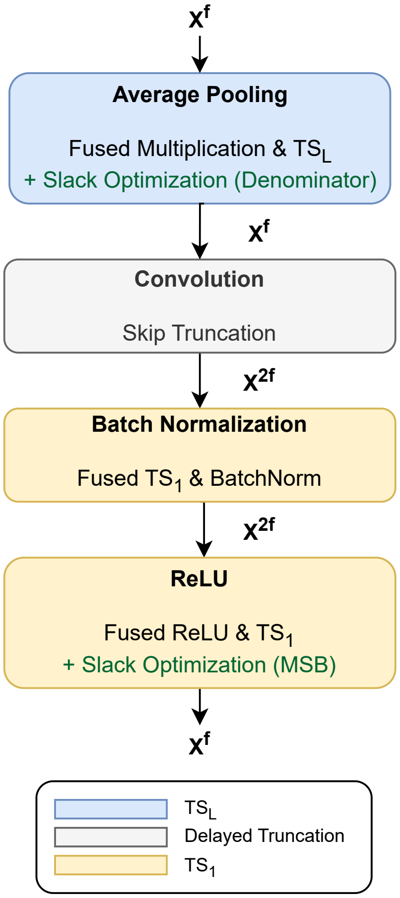


Figure 13: Mixed truncation workflow

While our evaluation focuses on secure CNN inference, we also discuss secure inference of Transformer architectures in §E.

6.1 PPML-specific Optimizations

This section evaluates the truncation optimizations proposed for PPML inference in §5. We categorize the evaluation based on the nature of the optimizations: slack-related, plaintext training-related, and performance-related.

6.1.1 Slack-related optimizations. Our slack-related optimization leverages non-negativity and reduces the number of fractional bits in the denominator during average pooling. Delaying truncation not only lowers communication complexity by fusing ReLU and truncation primitives but also reduces truncation failures by ensuring non-negativity through ReLU. Additionally, layers following the activation function, such as average pooling, benefit from this non-negativity, eliminating the extra bit of slack required by $TS_{\{1\}}$ and $TE_{\{1\}}$. Reducing the number of fractional bits during average pooling further decreases the likelihood of overflows and truncation failures. As shown in Table 5, these optimizations significantly enhance the accuracy of $TS_{\{1\}}$ and $TS_{\{Mix\}}$, bringing them closer to their deterministic counterparts and minimizing accuracy loss compared to plaintext inference.

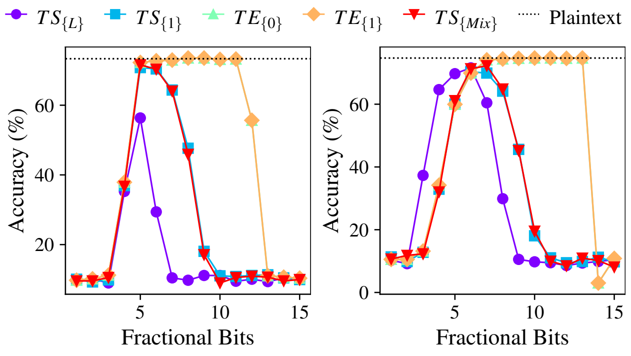
Table 5: Truncation accuracy in % for VGG-16 on CIFAR-10 with bitlength $\ell = 32$ and $t = 5$ fractional bits. Plaintext Accuracy: 81.74%.

	$\neg$OPT$^{\text{MSB}}$		OPT$^{\text{MSB}}$	
Scheme	$\neg$OPT$^{\text{AVG}}$	OPT$^{\text{AVG}}$	$\neg$OPT$^{\text{AVG}}$	OPT$^{\text{AVG}}$
$TS_{\{L\}}$	11.62	18.36	10.75	18.36
$TS_{\{1\}}$	51.47	81.05	66.31	81.05
$TE_{\{0\}}$	80.76	80.86	80.76	80.86
$TE_{\{1\}}$	80.76	80.86	80.76	80.86
$TS_{\{Mix\}}$	61.04	66.90	71.88	79.49

OPT^{MSB}: Exploiting non-negativity during pooling and ReLU layers. Includes delayed truncation (cf. §5).
OPT^{AVG}: Slack-related optimizations to the denominator during average pooling (cf. §5).

6.1.2 Plaintext-training-related optimizations. Techniques such as dropout, weight decay, and weight clipping can help reduce the magnitude of weight, lowering the probability of overflows and truncation failures. While dropout and weight clipping generally improve PPML inference performance as long as plaintext accuracy remains high, weight decay may require more fractional bits to accurately represent smaller weights, potentially reducing accuracy.

To study this trade-off, we train multiple models on CIFAR-10 dataset with the ADAMW optimizer [31] with a weight decay hyperparameter of 0.03 and compare them to models trained with the ADAM optimizer, which does not apply weight decay. Figure 16 presents the accuracy of different truncation schemes on CIFAR-10 with ResNet50 at a bitlength of 32. Additional results for various architectures and bitlengths are presented in §G (cf. Figure 20).



(a) ADAM, $\ell = 32$ **(b) ADAMW, $\ell = 32$**

Figure 14: Accuracy of truncation approaches with ResNet50 on CIFAR-10. Weight decay for ADAMW is set to 0.03.

We observe that models struggle to match plaintext accuracy when fractional bits are limited. Without weight decay, 5 fractional bits are sufficient for most truncation methods to achieve near-plaintext accuracy. However, with weight decay, accuracy declines noticeably at lower fractional bit settings. This decay is mitigated as the number of fractional bits increases. More importantly, weight decay reduces truncation failures: with weight decay and 6 fractional bits, $TS_{\{L\}}$ nearly reaches plaintext accuracy, whereas without weight decay, it fails to do so at any number of fractional bits.

6.1.3 Performance-related optimizations. We analyze the impact on communication complexity when delaying the truncation of one layer's output until the next layer, where it can be fused with another operation. Table 6 presents the reduction in communication complexity for a full forward pass using different truncation schemes on VGG16 with ImageNet (preprocessing + online). Additional results for other models, datasets, and bitlengths are presented in §H.

Table 6: Trio and Quad: Reduction in communication complexity of different truncation schemes for VGG16 on ImageNet when delaying truncation.

		$\ell = 32$			$\ell = 64$	
Scheme	¬ D	D	Δ	¬ D	D	Δ
			3PC			
$TS_{\{L\}}$	773.3	827.5	-6.55%	1554	1662	-6.52%
$TS_{\{1\}}$	1044	995.3	4.93%	2095	1992	5.17%
$TE_{\{0\}}$	1305	1039	25.62%	2627	2091	25.68%
$TE_{\{1\}}$	1404	1192	17.79%	2842	2412	17.80%
$TS_{\{Mix\}}$	1044	995.3	4.93%	2095	1992	5.17%
			4PC			
$TS_{\{L\}}$	1331	1440	-7.53%	2674	2891	-7.50%
$TS_{\{1\}}$	1819	1719	5.83%	3649	3441	6.05%
$TE_{\{0\}}$	2182	1702	28.19%	4392	3424	28.25%
$TE_{\{1\}}$	2421	2050	18.12%	4900	4149	18.11%
$TS_{\{Mix\}}$	1819	1719	5.83%	3649	3441	6.05%

¬D: Total communication in MB if not delaying truncation.
D: Total communication in MB if delaying truncation.
Δ: Percentage reduction in communication complexity.

As shown in the table, our approach significantly reduces total communication, particularly for exact truncation schemes. This is mainly because fusing these schemes with the ReLU operation eliminates most or all truncation-related overhead, which can account for over 25% of the total communication complexity in a forward pass. Stochastic truncation schemes also benefit from delayed truncation. An exception is $TS_{\{L\}}$, which can already be fused into each multiplication in Trio and Quad at no additional cost. However, to fully eliminate $TS_{\{L\}}$'s overhead when delaying truncation, a fused bit injection scheme similar to $TS_{\{1\}}$ could be designed.

Although we do not empirically verify round complexity, our optimizations likely have an even greater impact since exact truncation primitives often require multiple rounds, whereas linear layers, average pooling, and Batch Normalization can be computed in 0 to 1 communication rounds. Notably, reducing the communication complexity of $TS_{\{1\}}$ and $TS_{\{Mix\}}$ from 1 to 0 rounds by fusing truncation with other layers is a significant improvement, effectively halving the round complexity of these layers.

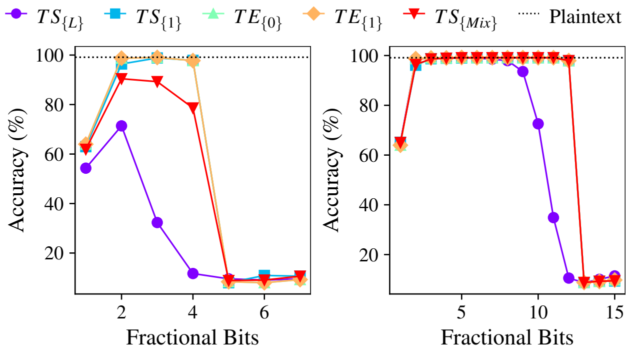
The overall impact on round complexity also depends on the boolean circuit used for sign bit extraction. This complexity ranges from $\log_4(\ell)$ rounds when using multi-input scalar products with a parallel prefix adder to $\ell - 1$ rounds when using a ripple carry adder.

6.2 Comparison of Truncation Approaches

This section compares different truncation approaches across various datasets in terms of accuracy and performance. For completeness, the communication complexity of all evaluated models and datasets is provided in §H, with tables 12 and 13 detailing the results for 3PC and 4PC, respectively.

6.2.1 Accuracy Comparison. To compare the accuracy across truncation schemes, we apply all the optimizations considered in this work and set the precision loss threshold for average pooling (cf. §5) to 0. This ensures that fractional bits are only reduced when it does not impact approximation accuracy. This section presents results for a subset of datasets and models, with additional results for other architectures available in §G (cf. Figure 19).

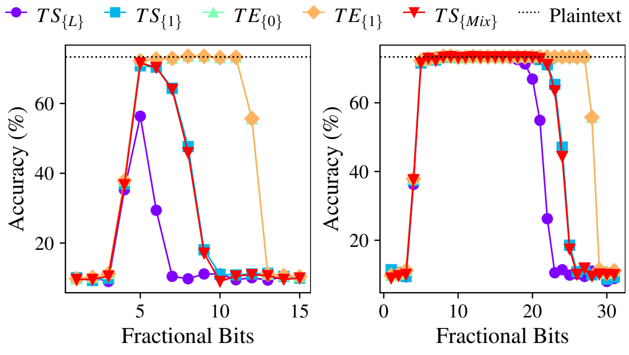
MNIST. We train a plaintext PyTorch model using the ADAM optimizer on a modified LeNet5 architecture, replacing max pooling with average pooling and using only ReLU activations for an MPC-friendly architecture. As shown in Figure 15, most truncation schemes closely match the plaintext accuracy of over 99% with a bitlength of 16 and 3 fractional bits. Only $TS_{\{Mix\}}$ and $TS_{\{L\}}$ require a bitlength of 32 to reach plaintext accuracy.



(a) $\ell = 16$ **(b)** $\ell = 32$

Figure 15: Accuracy of truncation approaches with LeNet5 on MNIST.

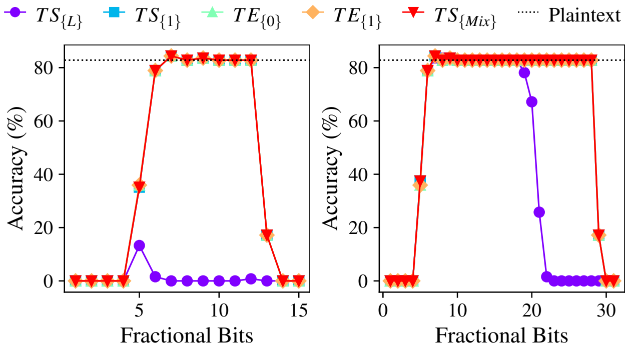
CIFAR-10. For CIFAR-10, we train PyTorch models with the ADAM optimizer on architectures such as ResNet50, VGG16, and AlexNet, replacing max pooling with average pooling. Figure 16 presents accuracy results for ResNet50. All truncation schemes except $TS_{\{L\}}$ nearly match plaintext accuracy with a bitlength of 32 and 5 fractional bits, while $TS_{\{L\}}$ requires a bitlength of 64. At a bitlength of 16, none of the truncation schemes achieve more than 40% accuracy.



(a) $\ell = 32$ **(b)** $\ell = 64$

Figure 16: Accuracy of truncation approaches with ResNet50 on CIFAR-10.

ImageNet. ImageNet contains over a million images, each sized 224x224x3, making full plaintext training consuming weeks. Instead, we use pre-trained PyTorch models of VGG16 and AlexNet, achieving over 80% and 60% plaintext accuracy, respectively. Both models employ max pooling, and accuracy is evaluated on 128 validation images. Figure 17 shows the accuracy results for VGG16. All truncation schemes except $TS_{\{L\}}$ match plaintext accuracy with a bitlength of 32 and 7 fractional bits, while $TS_{\{L\}}$ requires a bitlength of 64. At a bitlength of 16, none of the truncation schemes achieves more than 20% accuracy.



(a) $\ell = 32$ **(b)** $\ell = 64$

Figure 17: Accuracy of truncation approaches with VGG16 on ImageNet.

Recommended fixed-point ranges. From our evaluation of truncation schemes, we obtained the fractional ranges that result in 0%, 1%, and 5% accuracy loss compared to plaintext training. These results are detailed in §G (cf. Table 11). We further consolidated the findings to provide general recommendations based on dataset dimensions, with the suggested fractional ranges presented in Table 7. Each range ensures high accuracy for most models on the given dataset, with bold values indicating our recommended bitlength. This recommended bitlength and fractional range closely match the plaintext accuracy for most models, offering little to no additional accuracy gains when further increased.

Table 7: Recommended fixed-point range for truncation schemes.

Dataset	Scheme	Bitlength		
		$\ell = 16$	$\ell = 32$	$\ell = 64$
MNIST	$TS_{\{L\}}$	-	**5**	2-23
	$TS_{\{1\}}$	3	**6-11**	2
	$TE_{\{0\}}$	3-4	**5-11**	5
	$TE_{\{1\}}$	3-4	**5-11**	5
	$TS_{\{Mix\}}$	3-4	**5-11**	5
CIFAR-10	$TS_{\{L\}}$	-	6	**6-7**
	$TS_{\{1\}}$	-	**5-9**	8-12
	$TE_{\{0\}}$	-	**8-9**	8-12
	$TE_{\{1\}}$	-	**8-9**	8-12
	$TS_{\{Mix\}}$	-	**5-6**	8-12
ImageNet	$TS_{\{L\}}$	-	-	**8-18**
	$TS_{\{1\}}$	-	**10**	8-28
	$TE_{\{0\}}$	-	**8-12**	8-28
	$TE_{\{1\}}$	-	**8-12**	8-28
	$TS_{\{Mix\}}$	-	**10**	8-28

Table 8: Runtime (s) for different truncation schemes in MAN: 1 Gbit/s bandwidth, 2 ms latency.

Setting	Scheme	CIFAR-10				ImageNet	
		ResNet50		VGG-16		VGG-16	
		32	64	32	64	32	64
3PC	$TS_{\{L\}}$	5.07 ± 0.17	8.12 ± 0.00	2.18 ± 0.00	4.20 ± 0.00	8.16 ± 0.02	19.28 ± 0.31
	$TS_{\{1\}}$	5.68 ± 0.05	9.40 ± 0.30	2.23 ± 0.13	5.26 ± 0.30	8.75 ± 0.02	20.81 ± 0.02
	$TE_{\{0\}}$	21.95 ± 0.79	54.01 ± 0.14	5.23 ± 0.13	13.96 ± 0.82	9.47 ± 0.02	22.40 ± 0.29
	$TE_{\{1\}}$	11.92 ± 0.15	31.89 ± 0.80	3.03 ± 0.09	12.42 ± 1.62	9.34 ± 0.09	21.48 ± 0.08
	$TS_{\{Mix\}}$	5.23 ± 0.38	9.12 ± 0.01	2.22 ± 0.07	4.95 ± 0.00	8.63 ± 0.13	20.14 ± 0.02
4PC	$TS_{\{L\}}$	5.50 ± 0.10	12.72 ± 0.33	2.77 ± 0.14	8.25 ± 0.01	17.21 ± 0.04	41.73 ± 0.30
	$TS_{\{1\}}$	6.36 ± 0.04	12.20 ± 0.92	3.01 ± 0.41	8.63 ± 0.00	17.69 ± 0.06	42.30 ± 0.01
	$TE_{\{0\}}$	26.38 ± 1.34	39.09 ± 0.87	5.30 ± 0.04	13.78 ± 0.54	18.62 ± 0.28	44.33 ± 0.04
	$TE_{\{1\}}$	15.65 ± 0.00	21.39 ± 0.03	3.27 ± 0.14	7.23 ± 2.18	18.24 ± 0.07	43.44 ± 0.01
	$TS_{\{Mix\}}$	7.69 ± 0.02	11.53 ± 0.30	2.85 ± 0.00	8.42 ± 0.00	17.52 ± 0.04	42.21 ± 0.02

Additional results. We analyze the impact of truncating operands before multiplication versus truncating their product afterward, as proposed by Bicoptor 2.0 [51]. We find that truncation prior to multiplication is equivalent to a regular truncation approach but with half the number of fractional bits (cf. §B).

We also evaluate the common optimization of replacing MaxPooling with AveragePooling [45] and find that, given the significant reduction in communication complexity, this trade-off is worthwhile without substantially affecting plaintext accuracy. Additional results are presented in §C.

While we have demonstrated that PPML using fixed-point arithmetic can achieve the same accuracy as plaintext inference for large models and datasets, we aim to find an indicator of the limits of fixed-point approximation. Given the small accumulated fixed-point error in the final layer of neural networks, we conclude that fixed-point arithmetic remains sufficient for secure inference, even for models larger than those evaluated in this work (cf. §D).

6.2.2 Performance Comparison. In terms of communication complexity, Table 6 confirms that, as expected, $TS_{\{L\}}$ achieves the lowest communication complexity among all approaches at the same bitlength. However, when considering bitlengths based on the accuracy achieved by different truncation schemes, we arrive at a different conclusion.

As shown in Table 11 in §G, $TS_{\{Mix\}}$ and $TS_{\{1\}}$ exactly match the plaintext accuracy of all ImageNet models with bitlength $\ell = 32$ and 10 fractional bits. In contrast, $TS_{\{L\}}$ results in an accuracy loss of over 5% at $\ell = 32$ across all fractional bit settings. While increasing the bitlength to 64 mitigates this accuracy loss, $TS_{\{L\}}$ then incurs higher communication costs than all other truncation schemes. Thus, we conclude that $TS_{\{Mix\}}$ and $TS_{\{1\}}$ are the most efficient truncation schemes when balancing both accuracy and communication complexity.

For the case of runtime, we evaluate the end-to-end inference runtime (preprocessing + online phase) of different truncation schemes across three network settings:

- LAN: 25 Gbit/s bandwidth, 0.3 ms latency
- MAN: 1 Gbit/s bandwidth, 2 ms latency
- WAN: 200 Mbit/s bandwidth, 40 ms latency

Table 8 provides the runtime (in seconds) for different models in the MAN setting, while results for LAN and WAN are presented in §I (cf. Tables 14 and 15).

In line with communication complexity results, $TS_{\{L\}}$ achieves the lowest runtime among truncation schemes at the same bitlength. However, when using the recommended bitlength for accuracy, $TS_{\{L\}}$ has a higher runtime than other truncation schemes.

The results also highlight the advantage of $TE_{\{1\}}$ over $TE_{\{0\}}$ in parallelizing the computation of Boolean adders. Despite its higher communication complexity, $TE_{\{1\}}$ achieves a lower runtime than $TE_{\{0\}}$ in both MAN and WAN settings across all models and bitlengths. Additionally, for the VGG-16 model on ImageNet, the relative runtime differences between truncation schemes are smaller, as the MaxPooling layers present in the default architecture are responsible for the majority of the runtime.

6.3 Takeaways

Contrary to common intuition, $TS_{\{L\}}$ is not the most efficient stochastic truncation scheme when accounting for both accuracy and runtime. Deterministic truncation schemes achieve similar accuracy to stochastic ones but tend to be more reliable at higher fractional bit settings. Among them, $TE_{\{1\}}$ matches the accuracy of $TE_{\{0\}}$ while significantly reducing communication rounds. Therefore, $TS_{\{Mix\}}$, $TS_{\{1\}}$, and $TE_{\{1\}}$ are the most effective choices for efficient PPML inference.

For these schemes, a bitlength of $\ell = 32$ with 10 fractional bits serves as a strong baseline, closely matching plaintext accuracy for most models. In plaintext training, practitioners should consider replacing MaxPooling with AveragePooling and incorporating weight clipping and weight decay.

Our experiments show that, with proper configuration, 32-bit fixed-point PPML inference exactly matches the accuracy of plaintext floating-point accuracy in PyTorch for common CNN architectures on ImageNet. Additionally, our results suggest that 64-bit fixed-point arithmetic is a future-proof choice for MPC-based secure inference of large models and datasets due to its small accumulated deviation from floating-point calculations. Thus, practitioners can currently not expect any benefit from utilizing floating-point arithmetic for secure inference of CNNs.

Acknowledgements

We used ChatGPT4 to revise the text to correct typos and grammatical errors.

References

[1] Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. 2012. Secure computation on floating point numbers. *Cryptology ePrint Archive* (2012).

[2] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *ACM CCS*.

[3] Aner Ben-Efraim, Michael Nielsen, and Eran Omri. 2019. Turbospeedz: Double Your Online SPDZ! Improving SPDZ Using Function Dependent Preprocessing. In *ACNS*.

[4] Andreas Brüggemann, Robin Hundt, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2023. FLUTE: Fast and Secure Lookup Table Evaluations. In *IEEE S&P*.

[5] Ran Canetti. 2000. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology* (2000), 143–202.

[6] Octavian Catrina and Claudiu Dragulin. 2009. Multiparty Computation of Fixed-Point Multiplication and Reciprocal. In *Database and Expert Systems Applications*.

[7] Octavian Catrina and Amitabh Saxena. 2010. Secure Computation with Fixed-Point Numbers. In *FC*.

[8] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. 2019. ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction. In *CCSW@CCS*.

[9] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. 2020. Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. In *NDSS*.

[10] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. 2020. Secure Evaluation of Quantized Neural Networks. *PoPETs* (2020).

[11] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. 2021. Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security. In *USENIX Security*.

[12] Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. 2016. Confidential Benchmarking Based on Multiparty Computation. In *FC*.

[13] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. 2015. Automated synthesis of optimized circuits for secure computation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1504–1517.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. 4171–4186.

[15] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. 2020. Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits. In *CRYPTO*.

[16] Christopher Harth-Kitzerow. 2025. HPMPC: High-Performance Implementation of Secure Multiparty Computation (MPC) Protocols. https://github.com/chart21/hpmpc/. Accessed: 2025-02-26.

[17] Christopher Harth-Kitzerow, Ajith Suresh, Yonqing Wang, Hossein Yalame, Georg Carle, and Murali Annavaram. 2025. High-Throughput Secure Multiparty Computation with an Honest Majority in Various Network Settings. *PoPETs* (2025).

[18] Christopher Harth-Kitzerow, Yongqin Wang, Rachit Rajat, Georg Carle, and Murali Annavaram. 2025. PIGEON: A High Throughput Framework for Private Inference of Neural Networks using Secure Multiparty Computation. *PoPETs* (2025).

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. 770–778.

[20] Dan Hendrycks and Kevin Gimpel. 2016. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415* (2016).

[21] Zhicong Huang, Wen jie Lu, Cheng Hong, and Jiansheng Ding. 2022. Cheetah: Lean and Fast Secure Two-Party Deep Neural Network Inference. In *USENIX Security*.

[22] Neha Jawalkar, Kanav Gupta, Arkaprava Basu, Nishanth Chandran, Divya Gupta, and Rahul Sharma. 2024. Orca: FSS-based Secure Training and Inference with GPUs. In *IEEE S&P*.

[23] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *ACM CCS*.

[24] Marcel Keller, Peter Scholl, and Nigel P. Smart. 2013. An Architecture for Practical Actively Secure MPC with Dishonest Majority. In *ACM CCS*.

[25] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. 2021. SWIFT: Superfast and Robust Privacy-Preserving Machine Learning. In *USENIX Security*.

[26] Nishat Koti, Shravani Mahesh Patil, Arpita Patra, and Ajith Suresh. 2023. MPClan: Protocol Suite for Privacy-Conscious Computations. *Journal of Cryptology* (2023).

[27] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. 2022. Tetrad: Actively Secure 4PC for Secure Training and Inference. In *NDSS*.

[28] Yun Li, Yufei Duan, Zhicong Huang, Cheng Hong, Chao Zhang, and Yifan Song. 2023. Efficient 3PC for Binary Circuits with Application to Maliciously-Secure DNN Inference. In *USENIX Security*.

[29] Yehuda Lindell. 2020. Secure Multiparty Computation (MPC). ePrint Archive. https://eprint.iacr.org/2020/300

[30] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. 2015. Efficient Constant Round Multi-party Computation Combining BMR and SPDZ. In *CRYPTO*.

[31] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *ICLR*.

[32] Zoltán Ádám Mann, Christian Weinert, Daphnee Chabal, and Joppe W. Bos. 2024. Towards Practical Secure Neural Network Inference: The Journey So Far and the Road Ahead. *ACM Comput. Surv.* (2024).

[33] Payman Mohassel and Peter Rindal. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *ACM CCS*.

[34] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*.

[35] Lucien KL Ng and Sherman SM Chow. 2023. SoK: Cryptographic Neural-Network Computation. In *IEEE S&P*.

[36] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. 2024. Bolt: Privacy-preserving, accurate and efficient inference for transformers. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 4753–4771.

[37] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In *USENIX Security*.

[38] Arpita Patra and Ajith Suresh. 2020. BLAZE: Blazing Fast Privacy-Preserving Machine Learning. In *NDSS*.

[39] Deevashwer Rathee, Anwesh Bhattacharya, Rahul Sharma, Divya Gupta, Nishanth Chandran, and Aseem Rastogi. 2022. Secfloat: Accurate floating-point meets secure 2-party computation. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 576–595.

[40] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. 2021. Sirnn: A math library for secure rnn inference. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1003–1020.

[41] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International journal of computer vision* 115 (2015), 211–252.

[42] Manuel B. Santos, Dimitris Mouris, Mehmet Ugurbil, Stanislaw Jarecki, José Reis, Shubho Sengupta, and Miguel de Vega. 2024. Curl: Private LLMs through Wavelet-Encoded Look-Up Tables. In *CAMLIS*.

[43] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*.

[44] Ajith Suresh. 2021. *MPCLeague: Robust MPC Platform for Privacy-Preserving Machine Learning*. Ph. D. Dissertation. Indian Institute of Science (IISc), Bangalore. https://arxiv.org/abs/2112.13338.

[45] Sijun Tan, Brian Knott, Yuan Tian, and David J. Wu. 2021. CryptGPU: Fast Privacy-Preserving Machine Learning on the GPU. In *IEEE S&P*.

[46] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[47] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. 2021. Falcon: Honest-Majority Maliciously Secure Framework for Private Deep Learning. *PoPETs* (2021).

[48] Yongqin Wang, G Edward Suh, Wenjie Xiong, Benjamin Lefaudeux, Brian Knott, Murali Annavaram, and Hsien-Hsin S Lee. 2022. Characterization of mpc-based private inference for transformer-based models. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 187–197.

[49] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. 2022. Piranha: A GPU Platform for Secure Computation. In *USENIX Security*.

[50] Martin Zbudila, Erik Pohle, Aysajan Abidin, and Bart Preneel. 2024. MaSTer: Maliciously Secure Truncation for Replicated Secret Sharing Without Pre-processing. In *CANS*.

[51] Lijing Zhou, Qingrui Song, Su Zhang, Ziyu Wang, Xianggui Wang, and Yong Li. 2023. Bicoptor 2.0: Addressing Challenges in Probabilistic Truncation for Enhanced Privacy-Preserving Machine Learning. *CoRR arXiv* (2023). https://doi.org/10.48550/arXiv.2309.04909

A Merging Truncation and Bit Injection

When expanding the equation from §5, we obtain the equation below. Each $P_i \in \mathcal{P}_{\mathsf{m}_x}$ can compute an additive share of y by holding the input-dependent shares in the clear and obtaining $[\cdot]$-sharings of the relevant input-independent shares from $\mathcal{P}_{\lambda_x}$.

$$y = (\mathsf{m}_x + \lambda_x - 2\mathsf{m}_x\lambda_x)(\mathsf{m}_o - \lambda_o + \mathsf{m}_{km} - \lambda_{km} + \mathsf{m}_{kn} - \lambda_{kn} - 2(\mathsf{m}_{km}\mathsf{m}_{kn} - \mathsf{m}_{km}\lambda_{kn} - \mathsf{m}_{kn}\lambda_{km} + \lambda_{km}\lambda_{kn}))$$

$$\begin{aligned}
[y] = {} & \mathsf{m}_x\mathsf{m}_o - \mathsf{m}_x\left[\lambda_o\right] + \mathsf{m}_x\mathsf{m}_{km} - \mathsf{m}_x\left[\lambda_{km}\right] + \mathsf{m}_x\mathsf{m}_{kn} - \mathsf{m}_x\left[\lambda_{kn}\right] \\
& - 2(\mathsf{m}_x\mathsf{m}_{km}\mathsf{m}_{kn} - \mathsf{m}_x\mathsf{m}_{km}\left[\lambda_{kn}\right] - \mathsf{m}_x\mathsf{m}_{kn}\left[\lambda_{km}\right] + \mathsf{m}_x\left[\lambda_{km}\lambda_{kn}\right]) \\
& + \mathsf{m}_o\left[\lambda_x\right] - \left[\lambda_x\lambda_o\right] + \mathsf{m}_{km}\left[\lambda_x\right] - \left[\lambda_x\lambda_{km}\right] + \mathsf{m}_{kn}\left[\lambda_x\right] - \left[\lambda_x\lambda_{kn}\right] \\
& - 2(\mathsf{m}_{km}\mathsf{m}_{kn}\left[\lambda_x\right] - \mathsf{m}_{km}\left[\lambda_x\lambda_{kn}\right] - \mathsf{m}_{kn}\left[\lambda_x\lambda_{km}\right] + \left[\lambda_x\lambda_{km}\lambda_{kn}\right]) \\
& - 2(\mathsf{m}_x\mathsf{m}_o\left[\lambda_x\right] - \mathsf{m}_x\left[\lambda_x\lambda_o\right] + \mathsf{m}_x\mathsf{m}_{km}\left[\lambda_x\right] - \mathsf{m}_x\left[\lambda_x\lambda_{km}\right] + \mathsf{m}_x\mathsf{m}_{kn}\left[\lambda_x\right] - \mathsf{m}_x\left[\lambda_x\lambda_{kn}\right] \\
& + 4(\mathsf{m}_x\mathsf{m}_{km}\mathsf{m}_{kn}\left[\lambda_x\right] - \mathsf{m}_x\mathsf{m}_{km}\left[\lambda_x\lambda_{kn}\right] - \mathsf{m}_x\mathsf{m}_{kn}\left[\lambda_x\lambda_{km}\right] + \mathsf{m}_x\left[\lambda_x\lambda_{km}\lambda_{kn}\right])
\end{aligned}$$

$$\begin{aligned}
= {} & \left[\lambda_x\right](\mathsf{m}_o + \mathsf{m}_{km} + \mathsf{m}_{kn} - 2\mathsf{m}_{km}\mathsf{m}_{kn} - 2\mathsf{m}_x\mathsf{m}_o - 2\mathsf{m}_x\mathsf{m}_{km} - 2\mathsf{m}_x\mathsf{m}_{kn} + 4\mathsf{m}_x\mathsf{m}_{km}\mathsf{m}_{kn}) \\
& + \left[\lambda_o\right](-\mathsf{m}_x) + \left[\lambda_{km}\right](-\mathsf{m}_x + 2\mathsf{m}_x\mathsf{m}_{kn}) + \left[\lambda_{kn}\right](-\mathsf{m}_x + 2\mathsf{m}_x\mathsf{m}_{km}) + \left[\lambda_{km}\lambda_{kn}\right](-2\mathsf{m}_x) + \left[\lambda_x\lambda_o\right](-1 + 2\mathsf{m}_x) \\
& + \left[\lambda_x\lambda_{km}\right](-1 + 2\mathsf{m}_{kn} + 2\mathsf{m}_x - 4\mathsf{m}_x\mathsf{m}_{kn}) + \left[\lambda_x\lambda_{kn}\right](-1 + 2\mathsf{m}_{km} + 2\mathsf{m}_x - 4\mathsf{m}_x\mathsf{m}_{km}) + \left[\lambda_x\lambda_{km}\lambda_{kn}\right](-2 + 4\mathsf{m}_x) \\
& + \mathsf{m}_x(\mathsf{m}_o + \mathsf{m}_{km} + \mathsf{m}_{kn} - 2\mathsf{m}_{km}\mathsf{m}_{kn})
\end{aligned}$$

Observe that $\mathcal{P}_{\mathsf{m}_x}$ require the following additive shares to completely evaluate the equation:

$$\begin{aligned}
& \left[\lambda_x\right], \left[\lambda_o\right], \left[\lambda_{km}\right], \left[\lambda_{kn}\right], \left[\lambda_{km}\cdot\lambda_{kn}\right], \\
& \left[\lambda_x\cdot\lambda_o\right], \left[\lambda_x\cdot\lambda_{km}\right], \left[\lambda_x\cdot\lambda_{kn}\right], \left[\lambda_x\cdot\lambda_{km}\cdot\lambda_{kn}\right]
\end{aligned}$$

Out of these terms $\mathcal{P}_{\mathsf{m}_x}$ already hold $\left[\lambda_o\right]$, $\left[\lambda_{km}\right]$ and $\left[\lambda_{kn}\right]$ but not any of the cross-terms and not $\left[\lambda_x\right]^A$ since they only initially hold $\left[\lambda_x\right]^B$ in $\mathbb{Z}_2$. Thus, there are six remaining input-independent cross-terms that $\mathcal{P}_{\lambda_x}$ need to compute locally and share with $\mathcal{P}_{\mathsf{m}_x}$. $\mathcal{P}_{\mathsf{m}_x}$ can then proceed to compute $[y]$. Each $P_i \in \mathcal{P}_{\mathsf{m}_x}$ then samples λ_y^i, computes $\mathsf{M}_i = y^i + \lambda_y^i$ and sends it to the other party $P_j \in \mathcal{P}_{\mathsf{m}_x}$. Finally the parties set $\mathsf{m}_y = \mathsf{M}_1 + \mathsf{M}_2$ and all parties hold consistent sharings of $[\![y]\!]$ according to the Trio sharing semantics.

While this approach accounts for all steps to construct a semi-honest 3PC protocol in Trio, the malicious 4PC protocol in Quad requires additional steps to verify correctness. Note that the sharing of input-independent terms can be trivially verified as both P_0 and P_3 can compute and verify-share the input-independent terms with P_1 and P_2. However, the online reconstruction of $[\![y]\!]$ needs to be secured against a malicious P_1 or P_2 who might send incorrect messages. Thus, $\mathcal{P}_{0,1,2}$ engage in a similar computation to compute $[\overline{y}]$ which is only equal to $[y]$ if both P_1 and P_2 honestly communicated their messages. To do so, $\mathcal{P}_{0,1,2}$ sets $\overline{\mathsf{m}}_x = \mathsf{m}_x^* + \lambda_x$ for each input share used in the long equation, while $\mathcal{P}_{1,2}$ set $\overline{\mathsf{m}}_x = \mathsf{m}_x + \lambda_x^*$. Note that the parties inherently hold $\left[\overline{\lambda}_x\right] = \lambda_x^* + \lambda_x$ where λ_x is held by P_0 and λ_x^* is held by $\mathcal{P}_{1,2}$. The parties proceed to evaluate the identical equation that computes $[y]$ but with these new shares to compute $[\overline{y}]$ while P_3 supplies the six input-dependent terms denoted by $\overline{\lambda}$.

After obtaining $[\overline{y}]$, $\mathcal{P}_{1,2}$ verify-send their masked share to P_0 such that it can obtain $\mathsf{m}_{\overline{y}}^* = \overline{y} + \overline{\lambda}_y$. Finally, $\mathcal{P}_{0,1,2}$ compare their views of $\mathsf{m}_y + \overline{\lambda}_y$ and $\overline{\mathsf{m}}_y + \lambda_y$ using Π_{CV} to verify the correctness of the computation. Note that only if both P_1 and P_2 honestly communicated M_1 and M_2 the verification will succeed. As $\mathsf{M}_{\{1,2\}}$ is computed non-interactively by both P_1 and P_2, the verification is secure against a malicious corruption by one of the two parties.

Note that the compare-views used are identical to the ones used by Quad's multiplication protocol [17] and the authors provide simulation-based security proofs for the protocol. Also, note that $\mathsf{M}_{\{1,2\}}$ is only used for verification and thus part of the constant-round communication [17]. Figure 18 summarizes the protocol for semi-honest Trio and malicious Quad. Finally, observe that we can combine some input-independent terms to further reduce the number of input-independent shares that need to be sent from $\mathcal{P}_{\lambda_x}$ to $\mathcal{P}_{\mathsf{m}_x}$ to five.

$$\begin{aligned}
[y] = {} & \left[\lambda_x\right](\mathsf{m}_o + \mathsf{m}_{km} + \mathsf{m}_{kn} - 2\mathsf{m}_{km}\mathsf{m}_{kn} - 2\mathsf{m}_x\mathsf{m}_o \\
& - 2\mathsf{m}_x\mathsf{m}_{km} - 2\mathsf{m}_x\mathsf{m}_{kn} + 4\mathsf{m}_x\mathsf{m}_{km}\mathsf{m}_{kn}) \\
& + \left[\lambda_{km}\right](-\mathsf{m}_x + 2\mathsf{m}_x\mathsf{m}_{kn}) + \left[\lambda_{kn}\right](-\mathsf{m}_x + 2\mathsf{m}_x\mathsf{m}_{km}) \\
& + \left[\lambda_x\lambda_{km}\right](-1 + 2\mathsf{m}_{kn} + 2\mathsf{m}_x - 4\mathsf{m}_x\mathsf{m}_{kn}) \\
& + \left[\lambda_x\lambda_{kn}\right](-1 + 2\mathsf{m}_{km} + 2\mathsf{m}_x - 4\mathsf{m}_x\mathsf{m}_{km}) \\
& + \left[\lambda_\mathrm{o} + 2\lambda_\mathrm{km}\lambda_\mathrm{kn}\right](-\mathsf{m}_x) \\
& + \left[\lambda_\mathrm{x}\lambda_\mathrm{o} + 2\lambda_\mathrm{x}\lambda_\mathrm{km}\lambda_\mathrm{kn}\right](-1 + 2\mathsf{m}_x) \\
& + \mathsf{m}_x(\mathsf{m}_o + \mathsf{m}_{km} + \mathsf{m}_{kn} - 2\mathsf{m}_{km}\mathsf{m}_{kn})
\end{aligned}$$

The total communication complexity of the protocol is thus five elements in the preprocessing phase and two elements in the online phase for Trio and ten elements in the preprocessing phase and three elements in the online phase for Quad. Since Bit Injection requires the same online complexity and two resp. four elements of communication in the preprocessing phase, the total overhead of fusing truncation and Bit Injection is three preprocessing elements for Trio and six for Quad. This overhead is identical to the preprocessing costs of one three-input multiplication and thus we achieve exactly the same overhead as fusing Batch Normalization and stochastic truncation as shown in §5.

Protocol $\Pi_{\mathsf{BitInj+TS}_{\{1\}}}(\llbracket x \rrbracket^B, \llbracket o \rrbracket, \llbracket m \rrbracket, \llbracket m \rrbracket, k) \rightarrow \llbracket y \rrbracket$

Preprocessing:

(1) $\mathcal{P}_{\lambda_x}$ subset share $\lambda_x, \lambda_x \cdot \lambda_o, \lambda_{km} \cdot \lambda_{kn} \lambda_x \cdot \lambda_{km} \lambda_x \cdot \lambda_{kn} \lambda_x \cdot \lambda_{km} \cdot \lambda_{kn}$ with $\mathcal{P}_{\mathsf{m}_x}$.

(2) P_3 shares $\overline{\lambda}_x, \overline{\lambda}_x \cdot \overline{\lambda}_o, \overline{\lambda}_{km} \cdot \overline{\lambda}_{kn}, \overline{\lambda}_x \cdot \overline{\lambda}_{km}, \overline{\lambda}_x \cdot \overline{\lambda}_{kn}, \overline{\lambda}_x \cdot \overline{\lambda}_{km} \cdot \overline{\lambda}_{kn}$ with $\mathcal{P}_{0,1,2}$.

Online:

(1) $\mathcal{P}_{\mathsf{m}_x}$ locally compute $[y]$ using their input shares and preprocessing material provided by $\mathcal{P}_{\lambda_x}$.

(2) $\mathcal{P}_{0,1,2}$ locally compute $[\overline{y}]$ using their input shares and preprocessing material provided by $\mathcal{P}_3$.

(3) Each party $P_i \in \mathcal{P}_{\mathsf{m}_x}$ samples λ_y^i with $\mathcal{P}_{\lambda_x}$, computes $\mathsf{M}_i = y^i + \lambda_y^i$ and sends it to the other party $P_j \in \mathcal{P}_{\mathsf{m}_x}$. $\mathcal{P}_{\mathsf{m}_x}$ set $\mathsf{m}_y = \mathsf{M}_1 + \mathsf{M}_2$.

(4) $\mathcal{P}_{1,2}$ sample $\overline{\lambda}_y$ with $\mathcal{P}_3$, compute $\mathsf{M}_{\{1,2\}} = [\overline{y}] + \overline{\lambda}_y$, and verify-send it to P_0. P_0 sets $\mathsf{m}_y^* = [\overline{y}] + \mathsf{M}_{\{1,2\}}$.

(5) $\mathcal{P}_{0,1,2}$ compare their views of $\mathsf{m}_y + \overline{\lambda}_y$ and $\overline{\mathsf{m}}_y + \lambda_y$ using Π_{CV}.

(6) P_0 sets $\mathsf{m}_y^* = \overline{\mathsf{m}}_y - \overline{\lambda}_y$ while all other parties set $\lambda_y^* = \overline{\lambda}_y$. All parties now hold a consistent sharing of $\llbracket y \rrbracket$.

Figure 18: Merged Bit Injection and Truncation in Quad. Steps 2 of the preprocessing phase and steps 2,4,5,6 of the online phase are omitted for semi-honest Trio.

B Truncation before Multiplication

Bicoptor 2.0 [51] proposed to utilize truncation before multiplication to reduce the probability of truncation failure. When multiplying $c = a \cdot b$, truncation can either be applied to a and b individually before the multiplication or to the result c after the multiplication. When truncating c after multiplication, t is set to k where k is the number of fractional bits used to represent a value. When parties instead truncate a and b prior to multiplication, t is set to $\frac{k}{2}$. For large absolute values of a and b, truncation before multiplication can significantly reduce the probability of truncation failure by producing intermediary products with k instead of $2k$ fractional bits. As truncation prior to multiplication requires two individual truncations of a and b instead of a single one of c, the communication overhead is increased by a factor of two. However, we find that truncating prior to multiplication can be implemented without any additional communication overhead compared to truncating c after the multiplication.

Our key observation to perform the optimization is that each plaintext value can be pre-truncated by $t = \frac{k}{2}$ without any communication overhead before entering the MPC protocol. As a result, all secret shares are already pre-truncated, and multiplying them produces shares with k fractional bits. From that point on, truncating prior-to-multiplication can be implemented as truncating after multiplication by half the number of fractional bits as truncating $\langle c \rangle$ after multiplication by $t = \frac{k}{2}$ produces a pre-truncated share $\langle (c)^t \rangle$ that can be used either as the next $\langle (a)^t \rangle$ or $\langle (b)^t \rangle$ for further multiplications. However, observe also that pre-truncating all plaintext values by $k/2$ bits is equivalent to representing all fixed-point values with $k/2$ fractional bits to begin with. Hence, truncating prior to multiplication is actually no different from simply using half the number of fractional bits and utilizing the standard truncation after multiplication approach.

C Replacing MaxPooling with AveragePooling

A common optimization in MPC to reduce communication complexity is to replace max pooling with average pooling [45]. Computing the maximum of n values requires computing $n - 1$ pairwise comparisons along a tree of height $\log_2(n)$. Each pairwise comparison requires on DReLU operation. Hence, maxpooling with common kernel sizes such as 3x3 can become the most expensive layer in PPML while average pooling only requires a single fixed-point truncation and is typically the cheapest layer in PPML. To evaluate whether this optimization affects the accuracy of ML models we train different ResNet models on CIFAR-10 with maxpooling and average pooling using various optimizers and compare the accuracy. The results are shown in table 9. On average, the models with maxpooling achieve 72.59% accuracy while the models with average pooling achieve 71.00% accuracy. Given the significant reduction in communication complexity, we replace average pooling with max pooling for all further evaluations except ImageNet-based models where we rely on the official pretrained PyTorch models that use max pooling.

Table 9: Accuracy in % for ResNet models on CIFAR-10 with max pooling and average pooling.

Model	Optimizer	MaxPool	AvgPool
ResNet50	ADAM	77.14	72.04
	ADAMW	77.39	74.86
	SGD	66.41	63.30
	SGDW	73.38	72.95
ResNet101	ADAM	74.55	76.82
	ADAMW	76.00	75.22
	SGD	65.30	64.04
	SGDW	73.50	71.23
ResNet152	ADAM	76.33	75.31
	ADAMW	74.69	73.22
	SGD	62.77	61.45
	SGDW	73.62	71.58

[a] SGDW refers to SGD with 0.03 weight decay.

D How far can FPA scale?

FPA enables parties to use 16-bit, 32-bit, or 64-bit integer arithmetic that has low overhead on modern hardware. While ring sizes of more than 64-bit are possible, they introduce significant computational overhead. To obtain an indicator for how much precision is lost when utilizing 64-bit fixed-point arithmetic in PPML, we investigate the outputs computed by the last layer of the VGG-16 model on ImageNet and calculate δ^f as the fixed-point deviation of each fixed-point value with respect to its plaintext floating-point value. Note that by the last layer, all errors from previous layers' fixed-point calculations accumulate. We then calculate δ^c as the minimum difference of two final class predictions in floating-point. Intuitively, the closer δ^f and δ^c are the higher the probability that the fixed-point errors cause swapping the likelihood of two classes compared to floating-point arithmetic. As only swapping the likelihood order of top predictions is relevant in practice we only consider the top 5 most likely predictions. Over multiple batches we observe that the minimum δ^c found for VGG-16 on ImageNet is 0.22 meaning that two classes in the top 5 predictions are at least separated by 0.22. The maximum δ^f for $TS_{\{1\}}$ when using a bitlength of 64 and 13 fractional bits is 0.012 meaning that the fixed-point representation of a class value is at most 0.012 off from the final layer's floating-point value in plaintext inference. Given this discrepancy, we can conclude that using 64-bit fixed-point arithmetic in PPML can most likely be used for even larger models and datasets without risking misclassification due to the accumulation of fixed-point errors.

E PPML Inference of Transformer Architectures

Transformer architectures [46] are becoming increasingly popular in language and vision applications. In contrast to PPML inference with CNN models, Transformer models such as BERT [14] require GeLU [20], Softmax and Tanh implementations [48]. These require evaluating exponential and trigonometric functions which introduces impractical overhead in MPC [40].
The GeLU function is defined as:

$$\text{GELU}(x) = \frac{1}{2}x \cdot \left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$$

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

The Softmax function is practically implemented as:

$$\text{Softmax}(x_i) = \frac{e^{x_i - x_{\max}}}{\sum_{k=1}^{n} e^{x_k - x_{\max}}}$$

The Tanh function is defined as:

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

To enable Transformer-based inference, state-of-the-art PPML implementations utilize fixed-point arithmetic and approximate each of these functions. For instance, BOLT [36] approximates GeLU and Tanh using polynomial approximations of degree five and four, respectively, across three intervals. Softmax is approximated using a routine that involves evaluating a degree-two polynomial and a secure function to compute the reciprocal. BOLT reports absolute floating-point errors of 6.59×10^{-3} for Tanh, 9.77×10^{-4} for GeLU, and 1×10^{-6} for $exp(x)$, which is used in the Softmax computation. Thus, the dominant source of error in PPML inference with Transformers stems from these function approximations, rather than from fixed-point representation itself. On the positive side, Transformer models are often shallower than ResNets; for instance, BERT [14] typically uses 12 to 24 layers.

Despite the added complexity of evaluating Transformer-based architectures, BOLT demonstrates accuracy comparable to PyTorch when evaluating BERT models using fixed-point arithmetic with a bit length of 37 and 12 fractional bits. These results suggest that similar FPA configurations can be used for both Transformers and CNNs to achieve high accuracy. However, more extensive experimentation is needed to evaluate the accuracy of private Transformer-based inference across a broader range of datasets and models.

F Benchmark

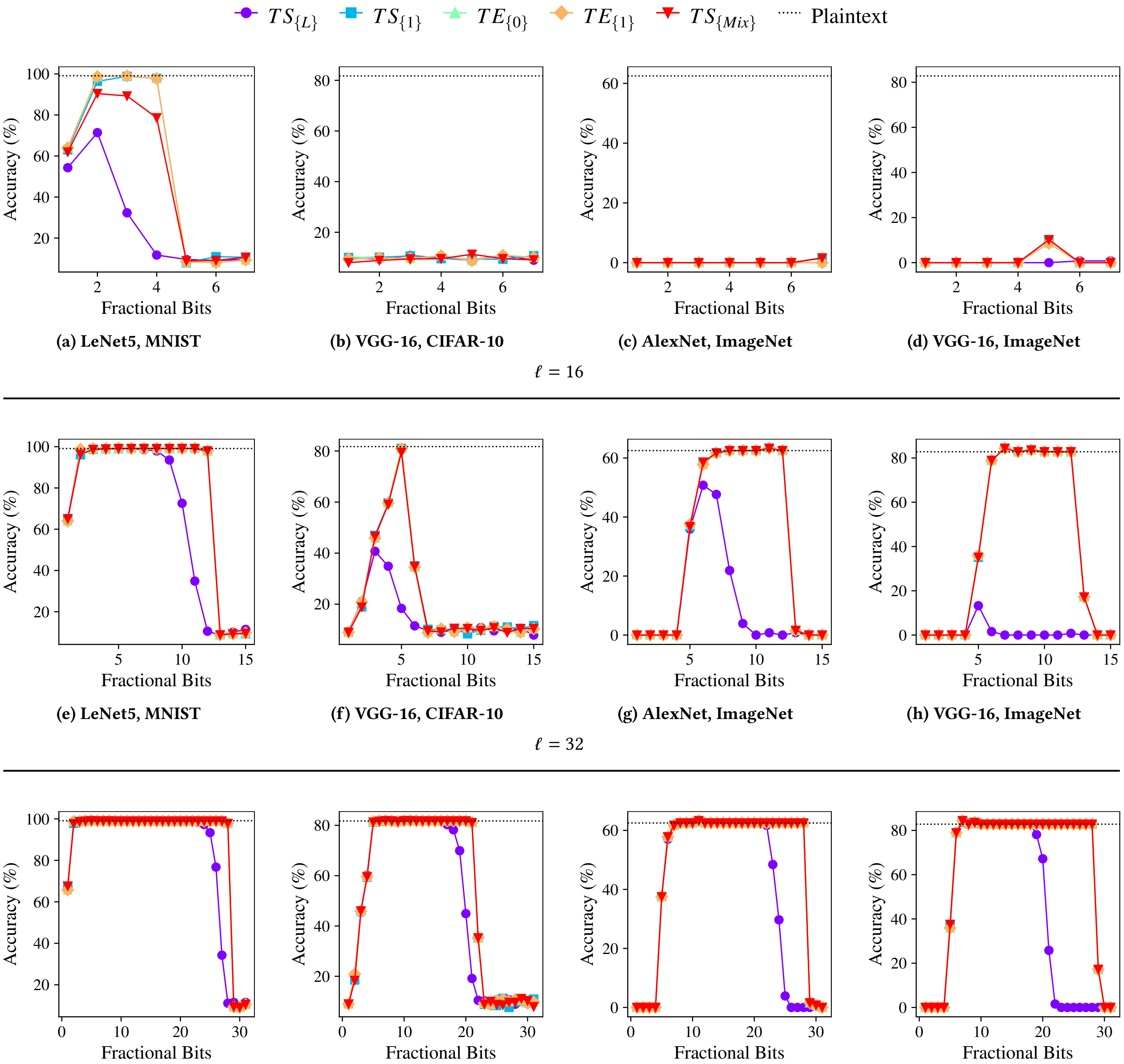
In sections §6.1 and §6.2 we identified $TS_{\{1\}}$ as the state-of-the-art truncation scheme offering the best trade-off between communication complexity and accuracy. Although this work focuses on improvements to various existing truncation schemes and how these truncation schemes perform on established benchmark datasets, we also evaluate how our implementation compares to state-of-the-art implementations.

The only existing implementation of $TS_{\{1\}}$ that we are aware of is provided by MP-SPDZ [23]. Table 10 compares our 3PC and 4PC implementation to MP-SPDZ's implementation using the same pre-trained VGG-16 model provided by PyTorch. The table shows that our implementation improves end-to-end inference runtime by more than one order of magnitude and communication complexity by 32%-57%.

Table 10: Runtime (s) and communication (MB) compared to MP-SPDZ for secure inference of VGG16 on ImageNet using 64-bit and $TS_{\{1\}}$ in a LAN setting

	Runtime (s)		Communication (MB)	
	3PC	4PC	3PC	4PC
MP-SPDZ [23]	736.05 ± 7.66	1814.01 ± 40.12	2945	7967
Ours	**15.80 ± 0.17**	**39.09 ± 0.16**	**1992**	**3441**

G Additional Accuracy Evaluation



(a) LeNet5, MNIST

(b) VGG-16, CIFAR-10

(c) AlexNet, ImageNet

(d) VGG-16, ImageNet

$\ell = 16$

(e) LeNet5, MNIST

(f) VGG-16, CIFAR-10

(g) AlexNet, ImageNet

(h) VGG-16, ImageNet

$\ell = 32$

(i) LeNet5, MNIST

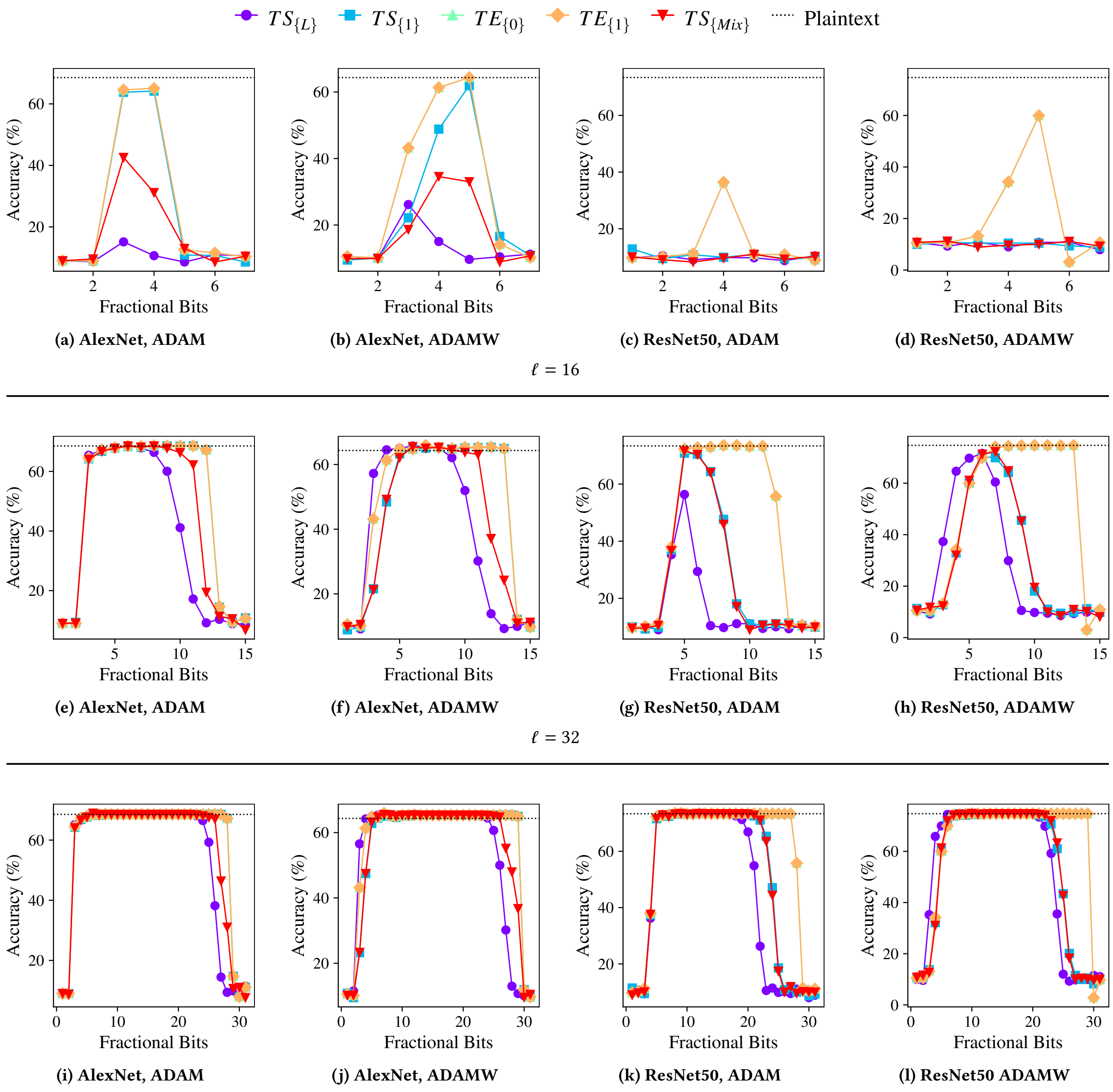
(j) VGG-16, CIFAR-10

(k) AlexNet, ImageNet

(l) VGG-16, ImageNet

$\ell = 64$

Figure 19: Accuracy of different truncation schemes on various models and datasets. Each row corresponds to a different Bitlength ℓ as indicated.



(a) AlexNet, ADAM **(b) AlexNet, ADAMW** **(c) ResNet50, ADAM** **(d) ResNet50, ADAMW**

$\ell = 16$

(e) AlexNet, ADAM **(f) AlexNet, ADAMW** **(g) ResNet50, ADAM** **(h) ResNet50, ADAMW**

$\ell = 32$

(i) AlexNet, ADAM **(j) AlexNet, ADAMW** **(k) ResNet50, ADAM** **(l) ResNet50 ADAMW**

$\ell = 64$

Figure 20: Accuracy of different truncation schemes for various models trained on CIFAR=10 with ADAMW and 0.03 weight decay or regular ADAM without weight decay. Each row corresponds to a different Bitlength ℓ as indicated.

Table 11: Ranges of fractional bits that introduce at most x% of accuracy loss compared to plaintext inference

Model	Plaintext Accuracy	Scheme	x=0%			x=1%			x=5%		
			$\ell = 16$	$\ell = 32$	$\ell = 64$	$\ell = 16$	$\ell = 32$	$\ell = 64$	$\ell = 16$	$\ell = 32$	$\ell = 64$
MNIST (28x28x1)											
LeNet	99.12%	$TS_{\{L\}}$	-	5	-	-	2-7	2-23	-	2-8	2-24
		$TS_{\{1\}}$	-	6-11	2	3	2-11	2-27	2-4	2-12	2-28
		$TE_{\{0\}}$	-	5-11	5	2-3	2-11	2-27	2-4	2-12	2-28
		$TE_{\{1\}}$	-	5-11	5	2-3	2-11	2-27	2-4	2-12	2-28
		$TS_{\{Mix\}}$	-	5-11	5	-	2-11	2-27	-	2-12	2-28
CIFAR-10 (32x32x3)											
AlexNet	68.55%	$TS_{\{L\}}$	-	6	11-21	-	6-7	5-23	-	3-8	3-24
		$TS_{\{1\}}$	-	8-9	8-27	-	5-11	6-27	-	3-12	4-28
		$TE_{\{0\}}$	-	8-11	8-27	-	5-11	5-27	-	4-12	4-28
		$TE_{\{1\}}$	-	8-11	8-27	-	5-11	5-27	-	4-12	4-28
		$TS_{\{Mix\}}$	-	6	6-23	-	5-8	6-25	-	4-10	3-26
AlexNet^{w}	64.33%	$TS_{\{L\}}$	-	4-8	5-24	-	4-8	4-24	-	4-9	4-24
		$TS_{\{1\}}$	-	5-13	5-29	-	5-13	5-29	4-5	4-13	4-29
		$TE_{\{0\}}$	5	5-13	5-29	5	5-13	5-29	4-5	4-13	4-29
		$TE_{\{1\}}$	5	5-13	5-29	5	5-13	5-29	4-5	4-13	4-29
		$TS_{\{Mix\}}$	-	4-9	4-26	-	4-10	4-26	-	4-11	4-26
ResNet50	73.34%	$TS_{\{L\}}$	-	-	8-9	-	-	6-18	-	-	5-19
		$TS_{\{1\}}$	-	-	8-9	-	-	6-21	-	5-7	5-23
		$TE_{\{0\}}$	-	8-9	8-9	-	6-11	6-27	-	5-11	5-27
		$TE_{\{1\}}$	-	8-9	8-9	-	6-11	6-27	-	5-11	5-27
		$TS_{\{Mix\}}$	-	-	8-9	-	-	6-22	-	5-7	5-23
ResNet50 ^{w}	74.71%	$TS_{\{L\}}$	-	-	9-10	-	-	6-20	-	6	6-21
		$TS_{\{1\}}$	-	-	6-7	-	6-7	6-22	-	6-7	6-23
		$TE_{\{0\}}$	-	-	-	-	7-13	7-29	-	7-13	7-29
		$TE_{\{1\}}$	-	-	-	-	7-13	7-29	-	7-13	7-29
		$TS_{\{Mix\}}$	-	-	6-7	-	-	6-22	-	5-7	6-23
VGG-16	81.74%	$TS_{\{L\}}$	-	-	6-7	-	-	5-16	-	-	5-18
		$TS_{\{1\}}$	-	-	10-12	-	5	5-21	-	5	5-21
		$TE_{\{0\}}$	-	-	11-12	-	-	5-21	-	5	5-21
		$TE_{\{1\}}$	-	-	11-12	-	-	5-21	-	5	5-21
		$TS_{\{Mix\}}$	-	-	10-12	-	-	5-21	-	5	5-21
ImageNet (224x224x3)											
AlexNet^{p}	62.50%	$TS_{\{L\}}$	-	-	11	-	-	8-21	-	-	7-22
		$TS_{\{1\}}$	-	10-11	25-26	-	8-12	8-28	-	7-12	7-28
		$TE_{\{0\}}$	-	11	11	-	8-12	8-28	-	7-12	7-28
		$TE_{\{1\}}$	-	11	11	-	8-12	8-28	-	7-12	7-28
		$TS_{\{Mix\}}$	-	10-11	25-26	-	8-12	8-28	-	7-12	7-28
VGG-16^{p}	82.81%	$TS_{\{L\}}$	-	-	7-9	-	-	7-18	-	-	6-18
		$TS_{\{1\}}$	-	7-10	7-9	-	7-12	7-28	-	6-12	6-28
		$TE_{\{0\}}$	-	7	7	-	7-12	7-28	-	6-12	6-28
		$TE_{\{1\}}$	-	7	7	-	7-12	7-28	-	6-12	6-28
		$TS_{\{Mix\}}$	-	7-10	7-9	-	7-12	7-28	-	6-12	6-28

^{w} Weight decay of 0.03 ^{p} Pretrained weights provided by PyTorch. Unmodified model architecture.

H Additional Evaluation of Communication Complexity

Table 12: Trio (3PC): Reduction in communication complexity of different truncation schemes for various models and datasets when delaying truncation.

Model	Scheme	$\ell = 16$			$\ell = 32$			$\ell = 64$		
		¬ D	D	Δ	¬ D	D	Δ	¬ D	D	Δ
					MNIST (28x28x1)					
LeNet	$TS_{\{L\}}$	0.144	0.157	-8.29%	0.291	0.317	-8.26%	0.589	0.641	-8.22%
	$TS_{\{1\}}$	0.222	0.212	4.97%	0.447	0.423	5.58%	0.900	0.853	5.52%
	$TE_{\{0\}}$	0.298	0.235	26.94%	0.605	0.476	27.22%	1.235	0.969	27.46%
	$TE_{\{1\}}$	0.332	0.282	17.76%	0.664	0.561	18.44%	1.366	1.152	18.62%
	$TS_{\{Mix\}}$	0.209	0.199	5.28%	0.422	0.398	5.93%	0.849	0.802	5.88%
					CIFAR-10 (32x32x3)					
ResNet18	$TS_{\{L\}}$	3.672	4.276	-14.13%	7.390	8.599	-14.06%	14.82	17.24	-14.01%
	$TS_{\{1\}}$	6.706	6.533	2.65%	13.46	13.06	3.00%	26.95	26.13	3.17%
	$TE_{\{0\}}$	9.519	8.410	13.19%	19.31	17.05	13.27%	38.90	34.34	13.28%
	$TE_{\{1\}}$	10.81	9.951	8.64%	21.52	19.75	8.96%	43.70	40.11	8.95%
	$TS_{\{Mix\}}$	6.672	6.499	2.66%	13.39	13.00	3.02%	26.83	26.00	3.18%
ResNet50	$TS_{\{L\}}$	5.794	6.705	-13.59%	11.66	13.49	-13.52%	23.40	27.04	-13.49%
	$TS_{\{1\}}$	10.36	10.07	2.97%	20.80	20.13	3.33%	41.67	40.25	3.53%
	$TE_{\{0\}}$	14.60	12.75	14.50%	29.62	25.84	14.60%	59.63	52.04	14.58%
	$TE_{\{1\}}$	16.54	15.10	9.58%	32.94	29.96	9.94%	66.85	60.85	9.86%
	$TS_{\{Mix\}}$	10.33	10.03	2.97%	20.74	20.06	3.35%	41.54	40.12	3.54%
VGG-16	$TS_{\{L\}}$	6.052	6.606	-8.39%	12.21	13.32	-8.32%	24.53	26.74	-8.27%
	$TS_{\{1\}}$	9.073	8.621	5.24%	18.25	17.25	5.82%	36.60	34.49	6.12%
	$TE_{\{0\}}$	11.92	9.257	28.81%	24.18	18.75	28.99%	48.71	37.72	29.14%
	$TE_{\{1\}}$	13.23	11.12	18.99%	26.42	22.09	19.62%	53.56	44.80	19.55%
	$TS_{\{Mix\}}$	8.822	8.372	5.38%	17.75	16.75	5.99%	35.60	33.49	6.30%
AlexNet	$TS_{\{L\}}$	0.267	0.291	-8.35%	0.538	0.586	-8.30%	1.080	1.178	-8.26%
	$TS_{\{1\}}$	0.403	0.383	5.19%	0.810	0.766	5.77%	1.624	1.531	6.09%
	$TE_{\{0\}}$	0.532	0.415	28.31%	1.080	0.840	28.57%	2.177	1.690	28.81%
	$TE_{\{1\}}$	0.591	0.498	18.69%	1.181	0.990	19.34%	2.398	2.007	19.48%
	$TS_{\{Mix\}}$	0.388	0.369	5.37%	0.781	0.737	6.01%	1.567	1.474	6.32%
					ImageNet (224x224x3)					
AlexNet	$TS_{\{L\}}$	28.15	29.14	-3.40%	56.86	58.84	-3.37%	114.3	118.2	-3.33%
	$TS_{\{1\}}$	33.10	32.28	2.52%	66.73	64.93	2.77%	134.0	130.2	2.91%
	$TE_{\{0\}}$	37.66	32.90	14.48%	76.25	66.53	14.61%	153.4	133.8	14.67%
	$TE_{\{1\}}$	39.77	35.99	10.50%	79.84	72.08	10.77%	161.2	145.5	10.79%
	$TS_{\{Mix\}}$	33.10	32.28	2.52%	66.73	64.93	2.77%	134.0	130.2	2.91%
VGG-16	$TS_{\{L\}}$	383.2	410.3	-6.60%	773.3	827.5	-6.55%	1554	1662	-6.52%
	$TS_{\{1\}}$	518.8	496.7	4.45%	1044	995.3	4.93%	2095	1992	5.17%
	$TE_{\{0\}}$	644.3	513.6	25.45%	1305	1039	25.62%	2627	2091	25.68%
	$TE_{\{1\}}$	701.8	598.4	17.28%	1404	1192	17.79%	2842	2412	17.80%
	$TS_{\{Mix\}}$	518.8	496.7	4.45%	1044	995.3	4.93%	2095	1992	5.17%

¬D: Total communication in MB if not delaying truncation.
D: Total communication in MB if delaying truncation.
Δ: Percentage reduction in communication complexity.

Table 13: Quad (4PC): Reduction in communication complexity of different truncation schemes for various models and datasets when delaying truncation.

Model	Scheme	$\ell = 16$			$\ell = 32$			$\ell = 64$		
		¬ D	D	Δ	¬ D	D	Δ	¬ D	D	Δ
MNIST (28x28x1)										
LeNet	$TS_{\{L\}}$	0.251	0.277	-9.42%	0.506	0.558	-9.36%	1.021	1.127	-9.37%
	$TS_{\{1\}}$	0.391	0.369	5.92%	0.784	0.736	6.50%	1.579	1.483	6.47%
	$TE_{\{0\}}$	0.495	0.381	29.92%	1.005	0.772	30.18%	2.051	1.572	30.44%
	$TE_{\{1\}}$	0.575	0.487	18.08%	1.149	0.968	18.74%	2.363	1.987	18.89%
	$TS_{\{Mix\}}$	0.369	0.347	6.31%	0.740	0.692	6.89%	1.491	1.394	6.90%
CIFAR-10 (32x32x3)										
ResNet18	$TS_{\{L\}}$	6.283	7.491	-16.13%	12.64	15.06	-16.06%	25.35	30.19	-16.01%
	$TS_{\{1\}}$	11.74	11.38	3.22%	23.56	22.75	3.55%	47.19	45.50	3.70%
	$TE_{\{0\}}$	15.62	13.61	14.77%	31.69	27.60	14.83%	63.85	55.60	14.84%
	$TE_{\{1\}}$	18.62	17.11	8.84%	37.05	33.95	9.14%	75.26	68.97	9.12%
	$TS_{\{Mix\}}$	11.68	11.32	3.22%	23.44	22.64	3.57%	46.96	45.27	3.73%
ResNet50	$TS_{\{L\}}$	9.922	11.74	-15.51%	19.97	23.61	-15.44%	40.05	47.35	-15.40%
	$TS_{\{1\}}$	18.15	17.52	3.58%	36.41	35.04	3.91%	72.94	70.06	4.11%
	$TE_{\{0\}}$	23.98	20.64	16.20%	48.66	41.85	16.28%	98.00	84.28	16.28%
	$TE_{\{1\}}$	28.50	25.96	9.78%	56.73	51.52	10.12%	115.2	104.7	10.04%
	$TS_{\{Mix\}}$	18.09	17.46	3.59%	36.30	34.92	3.93%	72.71	69.84	4.11%
VGG-16	$TS_{\{L\}}$	10.48	11.59	-9.56%	21.13	23.34	-9.49%	42.43	46.87	-9.45%
	$TS_{\{1\}}$	15.90	14.97	6.25%	31.97	29.93	6.82%	64.13	59.87	7.12%
	$TE_{\{0\}}$	19.83	15.02	32.05%	40.22	30.42	32.20%	81.01	61.23	32.31%
	$TE_{\{1\}}$	22.87	19.17	19.33%	45.66	38.07	19.94%	92.57	77.22	19.88%
	$TS_{\{Mix\}}$	15.46	14.53	6.44%	31.10	29.06	7.02%	62.38	58.12	7.33%
AlexNet	$TS_{\{L\}}$	0.463	0.511	-9.53%	0.932	1.029	-9.45%	1.871	2.066	-9.42%
	$TS_{\{1\}}$	0.707	0.666	6.17%	1.420	1.331	6.76%	2.848	2.660	7.06%
	$TE_{\{0\}}$	0.885	0.673	31.44%	1.795	1.363	31.68%	3.619	2.743	31.94%
	$TE_{\{1\}}$	1.023	0.860	19.00%	2.042	1.706	19.65%	4.145	3.461	19.77%
	$TS_{\{Mix\}}$	0.682	0.641	6.45%	1.370	1.280	7.03%	2.747	2.560	7.34%
ImageNet (224x224x3)										
AlexNet	$TS_{\{L\}}$	48.23	50.20	-3.92%	97.37	101.3	-3.89%	195.7	203.5	-3.88%
	$TS_{\{1\}}$	57.12	55.44	3.03%	115.2	111.5	3.29%	231.2	223.6	3.42%
	$TE_{\{0\}}$	63.43	54.83	15.69%	128.4	110.9	15.79%	258.3	223.0	15.84%
	$TE_{\{1\}}$	68.32	61.67	10.78%	137.1	123.5	11.00%	276.8	249.3	11.02%
	$TS_{\{Mix\}}$	57.12	55.44	3.03%	115.2	111.5	3.29%	231.2	223.6	3.42%
VGG-16	$TS_{\{L\}}$	659.8	714.0	-7.59%	1331	1440	-7.53%	2674	2891	-7.50%
	$TS_{\{1\}}$	903.8	858.0	5.34%	1819	1719	5.83%	3649	3441	6.05%
	$TE_{\{0\}}$	1077	841.0	28.05%	2182	1702	28.19%	4392	3424	28.25%
	$TE_{\{1\}}$	1211	1029	17.63%	2421	2050	18.12%	4900	4149	18.11%
	$TS_{\{Mix\}}$	903.8	858.0	5.34%	1819	1719	5.83%	3649	3441	6.05%

¬D: Total communication in MB if not delaying truncation.
D: Total communication in MB if delaying truncation.
Δ: Percentage reduction in communication complexity.

I Additional Runtime Evaluation

Table 14: Runtime (s) for different truncation schemes in LAN: 25 Gbit/s bandwidth, 0.3 ms latency.

Setting	Scheme	CIFAR-10				ImageNet	
		ResNet50		VGG-16		VGG-16	
		32	64	32	64	32	64
3PC	$TS_{\{L\}}$	3.35 ± 0.06	6.25 ± 0.05	2.72 ± 0.15	6.04 ± 0.03	6.51 ± 0.00	15.29 ± 0.05
	$TS_{\{1\}}$	4.17 ± 0.02	6.96 ± 0.38	3.23 ± 0.00	6.82 ± 0.00	6.90 ± 0.03	15.80 ± 0.17
	$TE_{\{0\}}$	14.59 ± 0.29	29.74 ± 0.32	6.74 ± 0.01	14.16 ± 0.02	6.75 ± 0.04	15.90 ± 0.10
	$TE_{\{1\}}$	8.72 ± 0.14	29.92 ± 0.01	3.56 ± 0.00	11.87 ± 0.01	7.23 ± 0.08	16.22 ± 0.27
	$TS_{\{Mix\}}$	3.87 ± 0.19	6.70 ± 0.08	3.11 ± 0.00	6.67 ± 0.12	6.86 ± 0.05	15.64 ± 0.04
4PC	$TS_{\{L\}}$	3.45 ± 0.03	6.41 ± 0.03	3.35 ± 0.02	6.67 ± 0.01	15.93 ± 0.08	38.34 ± 0.03
	$TS_{\{1\}}$	4.34 ± 0.00	7.01 ± 0.09	3.72 ± 0.01	7.04 ± 0.00	16.60 ± 0.02	39.09 ± 0.16
	$TE_{\{0\}}$	14.77 ± 0.53	15.48 ± 2.98	7.13 ± 0.04	14.75 ± 0.01	16.85 ± 0.24	39.34 ± 0.19
	$TE_{\{1\}}$	8.71 ± 0.08	5.24 ± 0.41	4.07 ± 0.01	13.34 ± 0.09	17.27 ± 0.03	40.30 ± 0.80
	$TS_{\{Mix\}}$	4.31 ± 0.00	5.13 ± 1.86	3.75 ± 0.00	6.99 ± 0.02	15.95 ± 0.33	39.15 ± 0.03

Table 15: Runtime (s) for different truncation schemes in WAN: 0.2 Gbit/s bandwidth, 40 ms latency.

Setting	Scheme	CIFAR-10			
		ResNet50		VGG-16	
		32	64	32	64
3PC	$TS_{\{L\}}$	69.05 ± 0.01	132.38 ± 0.00	20.95 ± 0.00	41.27 ± 0.04
	$TS_{\{1\}}$	75.43 ± 0.03	138.98 ± 0.11	22.46 ± 0.09	42.27 ± 0.00
	$TE_{\{0\}}$	282.40 ± 0.09	559.10 ± 0.92	54.03 ± 0.08	106.50 ± 0.27
	$TE_{\{1\}}$	153.10 ± 0.06	292.61 ± 0.15	29.33 ± 0.06	58.57 ± 0.00
	$TS_{\{Mix\}}$	72.44 ± 0.05	136.98 ± 0.12	22.31 ± 0.06	42.15 ± 0.12
4PC	$TS_{\{L\}}$	69.08 ± 0.00	132.53 ± 0.04	21.43 ± 0.02	41.87 ± 0.02
	$TS_{\{1\}}$	75.56 ± 0.07	138.77 ± 0.01	22.85 ± 0.02	43.02 ± 0.13
	$TE_{\{0\}}$	282.32 ± 0.06	559.33 ± 0.01	54.13 ± 0.04	106.65 ± 0.05
	$TE_{\{1\}}$	152.96 ± 0.33	292.49 ± 0.76	29.99 ± 0.04	56.51 ± 0.02
	$TS_{\{Mix\}}$	75.44 ± 0.01	138.74 ± 0.02	22.60 ± 0.03	42.69 ± 0.01