

# Privacy-Preserving Inference for Quantized BERT Models

Tianpei Lu<sup>1,3</sup>, Bingsheng Zhang<sup>1,3,\*</sup>, Lekun Peng<sup>1</sup>, Bowen Zheng<sup>1</sup>,  
Lichun Li<sup>2</sup>, Kui Ren<sup>1</sup>

<sup>1</sup>The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China, <sup>2</sup>Ant Group, China,

<sup>3</sup>Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security,  
 {lutianpei, bingsheng}@zju.edu.cn

## Abstract

With the increasing deployment of generative machine learning models in privacy-sensitive domains such as healthcare and personalized services, ensuring secure inference has become a critical challenge. Secure multi-party computation (MPC) enables privacy-preserving model inference but suffers from high communication and computation overhead. The main bottleneck lies in the expensive secure evaluation of floating-point operations. Quantization offers a promising solution by converting floating-point operations into lower-precision integer computations, significantly reducing overhead. However, existing MPC-based quantized inference methods either rely on public quantization parameters—posing privacy risks—or suffer from inefficiencies, particularly in handling nonlinear functions such as activations and softmax. In this work, we propose a fine-grained, layer-wise quantization scheme and support 1-bit weight fully connected layers in a secure setting. We design a multi-input lookup table protocol to evaluate softmax efficiently and securely. Furthermore, we use dual secret sharing schemes and perform precision conversions via lookup tables, eliminating truncation overhead entirely. Experimental evaluation on BERT-base models demonstrates that our approach achieves up to 8× speedup compared to Lu *et al.* (NDSS 25), 9× speedup compared to Gupta *et al.* (PETS 24) and 22× speedup compared to Knott *et al.* (NeurIPS 21).

## Introduction

With the widespread deployment of machine learning models in privacy-sensitive domains such as healthcare and personalized services, ensuring data confidentiality has emerged as a paramount concern. Privacy-preserving machine learning (PPML) techniques enable model inference while maintaining confidentiality, with secure multi-party computation (MPC) being the most popular approach. State-of-the-art (SOTA) PPML platforms, including Orca (Jawalkar et al. 2023), Falcon (Wagh et al. 2021), ABY<sup>3</sup> (Mohassel and Rindal 2018), and SecureML (Mohassel and Zhang 2017), predominantly leverage MPC frameworks. However, PPML techniques incur substantial communication overhead, especially for large-scale language

models. For instance, BERT-base with 12 transformer layers and over 110 million parameters requires billions of secure operations for a single inference. MPC-based inference can generate several gigabytes of communication, creating barriers for real-world deployment.

One of the key challenges in privacy-preserving machine learning is that most machine learning models are designed to operate using floating-point arithmetic, which incurs prohibitive overhead when implemented under MPC. The most efficient approach is to approximate these computations using fixed-point arithmetic. In fixed-point representation, a real number  $x$  is encoded as an integer  $\lfloor x \cdot 2^\ell \rfloor$  before being processed under MPC, where  $\ell$  denotes the number of fractional bits used to represent precision. However, after performing a multiplication  $x \cdot y$ , the result becomes  $x \cdot y \cdot 2^{2\ell}$ , and thus a truncation protocol is required to reduce it back to  $x \cdot y \cdot 2^\ell$ . This truncation step is costly. In current implementations, probabilistic truncation (Zhou et al. 2025; Mohassel and Zhang 2017; Mohassel and Rindal 2018; Rathee et al. 2020; Huang et al. 2022) is widely adopted, which necessitates reserving a large number of high-order bits as a tolerance margin to prevent truncation errors<sup>1</sup>. Specifically, for a truncation error bound of  $2^{-k}$ ,  $k$  extra bits are needed; the larger  $k$  is, the lower the probability of error. For multiplication of inputs with  $\ell_x$ -bit precision, at least  $2\ell_x$  bits are required to avoid overflow. When accounting for error tolerance in truncation, the overall bit-width expands to  $2\ell_x + k$ . For example, when  $\ell_x = 16$  and  $k = 32$ , the computation must be performed over a 64-bit ring. Moreover, in scenarios such as matrix summation, where intermediate results may exceed  $2\ell_x$ , an even larger bit-width may be necessary to ensure correctness and prevent overflow.

Quantization has emerged as a promising technique to enhance the performance of privacy-preserving machine learning (PPML) (Zhu et al. 2023; Luo et al. 2023; Zhu et al. 2021; Lang et al. 2023), particularly in distributed settings such as federated learning. However, its application in secure multiparty computation (MPC)-based PPML remains at an early stage. Several works, including those by Riazi *et*

<sup>1</sup>The truncation error is caused by the wrap bit. For example, in secret sharing,  $x_1 + x_2 \bmod 2^\ell = x$ . If a wrap occurs, then  $x_1 + x_2 = x + 2^\ell$ . When truncating by  $k$  bits, we compute  $x_1/2^k + x_2/2^k \bmod 2^\ell = x/2^k + 2^{\ell-k}$ , which introduces an error of  $2^{\ell-k}$ .

\*Bingsheng Zhang is the corresponding author.

Copyright © 2026, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

*al.*(Riazi et al. 2019), Agrawal *et al.*(Agrawal et al. 2019), and Keller *et al.* (Keller and Sun 2022), adopt traditional MPC protocols to evaluate quantized models. To optimize performance, these studies often rely on public, fixed quantization parameters—typically powers of two—which, while computationally efficient, can degrade model accuracy and risk leaking sensitive model information. Lu *et al.* (Lu et al. 2025) were the first to solve the truncation problem in quantized models through table lookup evaluation. While their approach ensures highly efficient online phase computation, it uses lookup tables to handle multiplication. This design results in a total of 256 bits of communication per multiplication gate (from 8-bit input to table lookup) in the offline phase, which makes the scheme challenging to deploy in practical scenarios. In this work, we address the limitations of Lu et al.’s method. Instead of using lookup tables for multiplication, which we find inefficient for matrix multiplication compared to the 2-out-of-3 protocol, we use lookup tables solely to handle truncation. Specifically, we truncate by discarding the lower bits (e.g., keeping the upper 4 bits from 16 bits) and then use a lookup table to expand the 4-bit value back to 16 bits. This avoids high-bit errors caused by truncation<sup>2</sup> (Zhou et al. 2025). In addition, by designing a multi-input lookup table protocol, we enable the efficient evaluation of functions such as Softmax. Overall, the contributions of this paper can be summarized as follows:

- We analyze the impact of weight and activation precision on model performance, and train the model with 1-bit weights and 4-bit activations. Under this quantization setting, privacy-preserving inference can be efficiently supported through secure multiplication combined with lookup table protocols.
- We design a multi-input lookup table protocol and, based on it, develop an efficient and systematic softmax evaluation protocol tailored for quantized models.
- We employ two types of secret sharing schemes to achieve layer-specific optimal implementations. To bridge different precisions across these schemes, we use lookup table evaluation to perform share conversion between different precisions, eliminating the overhead associated with truncation. As a result, our approach achieves up to 8× speedup compared to Lu *et al.* (NDSS 25), 9× speedup compared to Gupta *et al.* (PETS 24) and 22× speedup compared to Knott *et al.* (NeurIPS 21).

## Preliminaries

**Notations.** Let  $\mathcal{P} := \{P_0, P_1, P_2\}$  be the three MPC parties. We use  $\coloneqq$  to denote the definition. We use  $\mathbb{Z}_{2^\ell}$  to denote the set  $\{0, \dots, 2^\ell - 1\}$ . For a set  $S$ ,  $S^\ell$  indicates the  $\ell$  dimension vector space over the set  $S$ , namely,  $S^\ell := \{(s_0, s_1, \dots, s_{\ell-1}) \mid s_i \in S\}$ . We encode value  $x \in [-2^{\ell-1}, 2^{\ell-1})$  to the ring  $\mathbb{Z}_{2^\ell}$ . Here, for  $x \in [0, 2^\ell)$ , positive values remain unchanged, while negative values  $x$  are

<sup>2</sup>When truncating while also reducing the modulus to  $2^{\ell-k}$ , no truncation error occurs. If a wrap happens so that  $x_1 + x_2 = x + 2^\ell$ , then  $x_1/2^k + x_2/2^k \bmod 2^{\ell-k} = x$ , eliminating the error term.

encoded as  $2^\ell + x \in [2^{\ell-1}, 2^\ell]$ .  $\ell$  is the bit length of an element in  $\mathbb{Z}_{2^\ell}$ .  $x||y$  refers to the concatenation of  $x$  and  $y$ . We use  $\text{trc}(x, k)$  to denote the truncation of  $x$  to its first  $k$  bits.

**Two Party Additive Secret Sharing.** We denote two party additive secret sharing over ring  $\mathbb{Z}_{2^\ell}$  as  $\llbracket x \rrbracket^\ell := (\llbracket x \rrbracket_1^\ell, \llbracket x \rrbracket_2^\ell)$ . It holds that  $\llbracket x \rrbracket_1^\ell + \llbracket x \rrbracket_2^\ell \bmod 2^\ell = x$ . The party  $P_1$  holds  $\llbracket x \rrbracket_1^\ell$  and  $P_2$  holds  $\llbracket x \rrbracket_2^\ell$ . It possesses additive homomorphic properties, meaning that  $\llbracket x + y \rrbracket^\ell = (\llbracket x \rrbracket_1^\ell + \llbracket y \rrbracket_1^\ell \bmod 2^\ell, \llbracket x \rrbracket_2^\ell + \llbracket y \rrbracket_2^\ell \bmod 2^\ell)$ . When party  $P$  wants to secret-share a value  $x$ , it first agrees with party  $P_1$  on the same seed  $s$ . Then, both  $P$  and  $P_1$  use  $s$  to generate the random share  $[x]_1$ . Party  $P$  computes  $[x]_2 = x - [x]_1$ , and sends  $[x]_2$  to party  $P_2$ . We use  $\Pi_{\text{share}}(x, P)$  to denote the secret share scheme.  $P_1$  and  $P_2$  can exchange  $[x]_1$  and  $[x]_2$  with each other to reveal the value  $x = [x]_1 + [x]_2$ .

**Secure Three Party Replicated Secret Sharing.** Replicated Secret Sharing (RSS) is a secret sharing scheme commonly used in secure multi-party computation. It requires at least three parties. Compared to two-party computation, its advantage lies in not relying on public-key cryptography, enabling more efficient secure computation. In our setting, we define the 3PC-RSS over ring  $\mathbb{Z}_{2^\ell}$  as  $\langle x \rangle^\ell := (\langle x \rangle_0^\ell, \langle x \rangle_1^\ell, \langle x \rangle_2^\ell)$ . The party  $P_{i-1}$  and  $P_{i+1}$  both hold  $\langle \cdot \rangle_i$ . It holds that  $\langle x \rangle_0 + \langle x \rangle_1 + \langle x \rangle_2 \bmod 2^\ell = x$ . Similarly, replicated secret sharing also possesses homomorphic properties, namely,  $\langle x + y \rangle^\ell = (\langle x \rangle_0^\ell + \langle y \rangle_0^\ell, \langle x \rangle_1^\ell + \langle y \rangle_1^\ell, \langle x \rangle_2^\ell + \langle y \rangle_2^\ell)$ . 3PC-RSS is well-suited for evaluating multiplication protocols, especially matrix multiplication. This is because the communication cost of replicated secret sharing depends only on the output dimension. When computing vector inner products, since the output is a scalar, the required communication cost is constant. Below, we present the inner product protocol under replicated secret sharing. For inner product  $z = \sum_{j=1}^N x_j \cdot y_j$  with two input vector  $\{\langle x_j \rangle\}_{j \in \mathbb{Z}_N}$ ,  $\{\langle y_j \rangle\}_{j \in \mathbb{Z}_N}$  and output inner product result  $\langle z \rangle$ , each party  $P_i$  perform

$$z_i = \sum_{j=0}^{N-1} \langle x_j \rangle_{i-1} \cdot \langle y_j \rangle_{i+1} + \langle x_j \rangle_{i+1} \cdot \langle y_j \rangle_{i-1} \\ + \langle x_j \rangle_{i+1} \cdot \langle y_j \rangle_{i+1} .$$

It holds that  $z_0 + z_1 + z_2 = z$  is the inner product result. Then  $P_i$  send  $z_i$  to  $P_{i+1}$  to recover the 3PC-RSS form.

**Lookup Table.** The lookup table  $\mathcal{T} \in \mathbb{Z}_{2^\ell}^{\ell'}$  for function  $f : \{0, 1\}^{\ell'} \rightarrow \{0, 1\}^\ell$  traverse all possible inputs of  $f$ . It accepts  $\ell'$  bits input and output  $\ell$  bits message. The lookup table  $\mathcal{T}$  can be viewed as a vector. The  $r^{\text{th}}$  item of lookup table  $\mathcal{T}$  stores the result of  $f$  with input  $r$ . We use  $\mathcal{T}(r)$  to denote the  $r^{\text{th}}$  entry of the lookup table.

**Oblivious Maximum Evaluation.** Oblivious Maximum Evaluation is used to identify the maximum value from a set of secret-shared values while preserving privacy. In our setting, it takes as input a vector of secret shares and outputs the secret share of the maximum value. Asharov et al. (Asharov et al. 2022) proposed a radix sort algorithm for the three-party computation setting, where the number

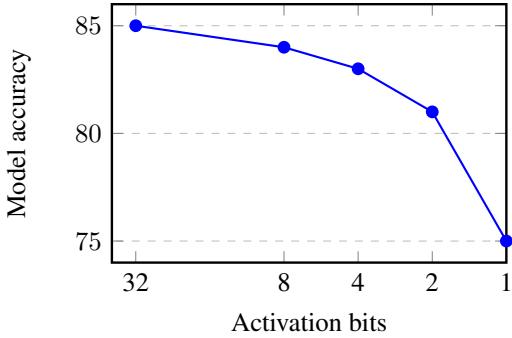


Figure 1: Model accuracy under 1-bit weight and different activation bit-widths

of sorting rounds depends only on the bit-length of the elements being sorted. In our work, we leverage Asharov et al.’s sorting algorithm to perform maximum evaluation: we first sort the input vector in ascending order and then select the last element as the maximum. We denote the above the maximum protocol as  $\Pi_{\max}$ .

**System Architecture and Threat Model.** Our framework involves three parties: the model owner  $P_0$ , the data owner  $P_1$ , and the computing assistant  $P_2$ . In this framework, the model owner publicly reveals the embedding parameters. The data owner first performs the embedding computation locally, and then quantizes the resulting embeddings into 4-bit values. These quantized embeddings are secret-shared among  $P_0$ ,  $P_1$ , and  $P_2$  using RSS. Our BERT inference framework ensures privacy in the semi-honest adversarial setting, meaning that as long as all parties follow the protocol, no party can obtain any information about the model weights or the input tokens.

## Our BERT Model Structure

The quantization process begins by applying low-bit quantization to the model parameters, including weights and activations (Activations refer to all intermediate variables within the model). Specifically, weights are binarized i.e., to 1 bit, by centering them around zero and applying a sign function, while activations are quantized to integers (e.g., 4-bit, 8-bit) based on their value distribution—either symmetric for values ranging from negative to positive, or asymmetric for strictly non-negative activations. Each quantized layer uses a layer-specific scaling factor  $\alpha$  to minimize quantization error, with the option to fine-tune both scaling factors using gradient-based optimization during training. After this initial quantization, we perform knowledge distillation to compensate for the loss of accuracy caused by low-bit representations. In this stage, a full-precision teacher model guides the training of the quantized student model. By aligning intermediate representations or output logits, the student learns to approximate the teacher’s behavior despite its reduced precision. This two-step procedure—quantization followed by distillation—enables efficient low-bit inference while preserving high model performance.

We conducted experiments under the framework pro-

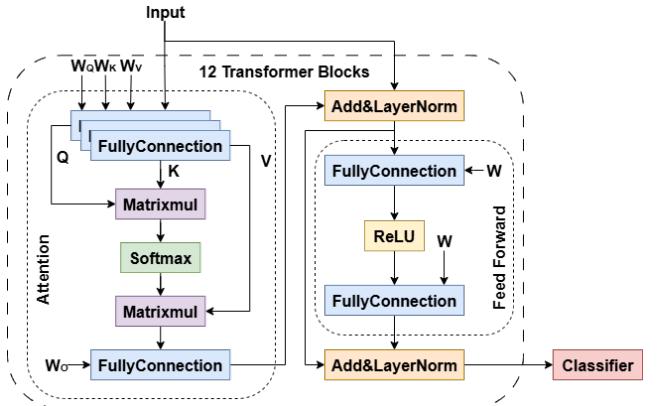


Figure 2: Overview of BitNet

posed by Liu *et al.* (Liu et al. 2022), evaluating model accuracy across various activation bits. As illustrated in Fig. 1, each weights are quantized to 1 bits and we benchmark different activation bits. Among these configurations, we found that 4-bit activation offers the best trade-off between model performance and efficiency. Based on this observation, we adopt the 4-bit activation setting as the foundation for our overall architecture.

**Model Structure.** Our protocol is divided into a linear computation part and a nonlinear part. The linear part includes fully connected layers and matrix multiplications. For the fully connected layers, the parameters come from within the model and are quantized to 1 bit. The quantized weights are represented as  $W \in \{-1, 1\}^{m \times n}$ , where  $m$  is the number of output features and  $n$  is the number of input features. The input activations are represented as  $x \in \{-8, -7, \dots, 6, 7\}^n$  for signed activations or  $x \in \{0, 1, \dots, 14, 15\}^n$  for unsigned activations. The output of the linear layer is computed as follows:

$$z = W \cdot x = \alpha_W \cdot \alpha_x \cdot W \cdot x \quad (1)$$

$$z' = \alpha_z \cdot \text{clip}\left(\lfloor \frac{z^i}{\alpha_z} \rfloor, -8, 7\right) \quad (2)$$

For the matrix multiplication, we use the same quantization scheme as in the fully connected layer. The only difference is that both inputs are activations, and we quantize both of them to 4 bits. For the nonlinear layers, including ReLU, Softmax, and others, these operators do not involve internal parameters; all the inputs are activations, which are directly quantized to 4 bits. Take Softmax layers as an example, we compute the quantized activations as follows:

$$z = \alpha_z \cdot \text{clip}\left(\lfloor \frac{\text{Softmax}(y \cdot \alpha_x)}{\alpha_z} \rfloor, 0, 15\right) \quad (3)$$

## Our New Technique

The core logic of our framework is to convert floating-point computations in transformer models into integer computations through model quantization. We use Replicated Secret Sharing (RSS) to evaluate multiplications, and utilize two-party additive secret sharing held by  $P_1$  and  $P_2$  to evaluate

lookup tables. We first introduce the lookup table evaluation protocol.

### Secure Lookup Table Evaluation.

In our designed multi-input lookup table protocol, we first introduce how to securely evaluate a lookup table. In the three-party setting, secure lookup table evaluation can be implemented quite simply (Lu et al. 2025). Specifically, for a lookup table  $\mathcal{T}$  held by  $P_0$  with  $\ell'$  bit input, which means it contains  $2^{\ell'}$  items,  $P_0$  first randomly generates an offset value  $\Delta$ , then left-shifts the table  $\mathcal{T}$  by  $\Delta$  positions and secret share the shifted table to  $P_1$  and  $P_2$ , namely,  $\llbracket \mathcal{T}' \rrbracket := (\llbracket \mathcal{T}(\Delta) \rrbracket, \dots, \llbracket \mathcal{T}(\Delta - 1) \rrbracket)$ . At the same time,  $\Delta$  is also secret-shared to  $P_1$  and  $P_2$ . Then, when  $P_1$  and  $P_2$  input the secret share  $\llbracket x \rrbracket$ , they only need to compute  $\llbracket x \rrbracket - \llbracket \Delta \rrbracket$ , and open it. After that,  $P_1$  and  $P_2$  locally take the  $(x - \Delta)$ -th entry from the secret-shared lookup table as the output. The correctness can be easily verified:  $\mathcal{T}'(x - \Delta) = \mathcal{T}(x - \Delta + \Delta) = \mathcal{T}(x)$ . Fig. 1 depicts this algorithm.

Algorithm 1: Lookup table evaluation  $\Pi_{\text{look}}$ .

---

**Input:** lookup table  $\mathcal{T} \in \mathbb{Z}_{2^{\ell'}}^{2^{\ell'}}$  provided by  $P_0$ , 2PC additive shared value  $\llbracket x \rrbracket^{\ell'}$  held by  $P_1$  and  $P_2$   
**Output:** shares of  $\mathcal{T}(x)$

- 1:  $P_0$  picks random  $\Delta \in \mathbb{Z}_{2^{\ell'}}$  and left-shifts  $\mathcal{T}$  by  $\Delta$  positions;
- 2:  $P_0$  invoke  $\llbracket \mathcal{T} \rrbracket^{\ell'} \leftarrow \Pi_{\text{share}}(\mathcal{T}, P_0)$  and  $\llbracket \Delta \rrbracket^{\ell'} \leftarrow \Pi_{\text{share}}(\Delta, P_0)$ ;
- 3:  $P_1$  and  $P_2$  calculate  $\llbracket \delta \rrbracket^{\ell'} = \llbracket x \rrbracket - \llbracket \Delta \rrbracket^{\ell'}$ , reveal  $\delta$  to each other;
- 4:  $P_1$  and  $P_2$  set output as  $\llbracket \mathcal{T}(\delta) \rrbracket^{\ell'}$ ;

---

In this work, we extend the single-input lookup table to support multiple inputs. Compared to single-input lookup tables, multi-input lookup tables are more complex. Below we intuitively explain the challenges of multi-input settings using a two-input example. For a total input of  $\ell'$  bits, it is divided into two parts  $x$  and  $y$ , each of  $\ell'/2$  bits, and secret-shared under  $\ell'/2$ -bit secret sharing. Directly concatenating the two secret shares incurs significant overhead. For example, one may first convert  $x$  and  $y$  from  $\ell'/2$ -bit secret shares to  $\ell'$ -bit secret shares, then compute  $x \cdot 2^{\ell'/2} + y$ , and finally evaluate a single-input lookup table. However, converting from  $\ell'/2$ -bit to  $\ell'$ -bit secret sharing is a nonlinear operation with high cost. We design a multi-input lookup table evaluation scheme whose overhead matches that of the single-input case. Our scheme is illustrated in Fig. 3.

We adopt a two- $\Delta$  approach. First, we generate the lookup table  $\mathcal{T}(x||y) = f(x, y)$ . For the inputs  $x$  and  $y$ , which need to be concatenated into  $x||y$ , we treat this as two indexing steps: the first uses  $x$  to index the higher  $\ell'/2$  bits, and the second uses  $y$  to index the lower  $\ell'/2$  bits. Based on this idea, during the generation of the lookup table, we group every  $2^{\ell'/2}$  small entries into blocks. A global offset  $\Delta \in [0, 2^{\ell'/2}]$  is generated for the outer blocks, and a shift is applied accordingly. Then, for each block, we apply a same

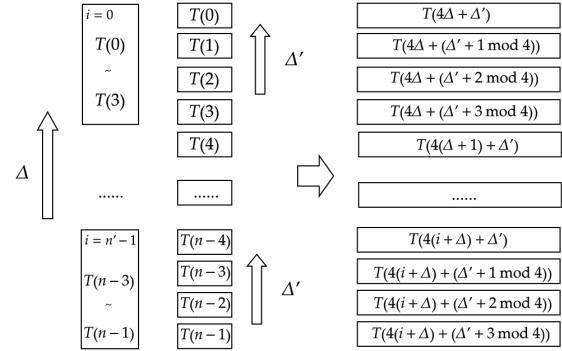


Figure 3: Lookup Table with Separate Inputs. (Take  $\ell' = \ell = 2$ )

second shift using a local offset  $\Delta'$ . Next, we open  $\delta = x - \Delta$  and  $\delta' = y - \Delta'$ , and directly output the result as the  $\delta$ -th block's  $\delta'$ -th entry.

Algorithm 2: Lookup table with separate inputs  $\Pi_{\text{look}}^{\ell'/2, \ell'/2}$ .

---

**Input:** lookup table  $\mathcal{T} \in \mathbb{Z}_{2^{\ell'}}^{2^{\ell'}}$  provided by  $P_0$ , 2PC additive shared values  $\llbracket x \rrbracket^{\ell'/2}$  and  $\llbracket y \rrbracket^{\ell'/2}$  held by  $P_1$  and  $P_2$   
**Output:** shares of  $\mathcal{T}(x||y)$

- 1:  $P_0$  picks random  $(\Delta, \Delta') \in \mathbb{Z}_{2^{\ell'/2}}^2$ ;
- 2:  $P_0$  left-shifts  $\mathcal{T}$  by  $2^{\ell'/2} \cdot \Delta$  positions;
- 3: For  $(i, j) \in \mathbb{Z}_{2^{\ell'/2}}^2$ ,  $P_0$  moves  $\mathcal{T}(i \cdot 2^{\ell'/2} + j)$  to position  $i \cdot 2^{\ell'/2} + (j - \Delta' \bmod 2^{\ell'/2})$  of  $\mathcal{T}$ ;
- 4:  $P_0$  invoke  $\llbracket \mathcal{T} \rrbracket^{\ell'} \leftarrow \Pi_{\text{share}}(\mathcal{T}, P_0)$ ,  $\llbracket \Delta \rrbracket^{\ell'} \leftarrow \Pi_{\text{share}}(\Delta, P_0)$  and  $\llbracket \Delta' \rrbracket^{\ell'} \leftarrow \Pi_{\text{share}}(\Delta', P_0)$ ;
- 5:  $P_1$  and  $P_2$  do
  - $\llbracket \delta \rrbracket^{\ell'/2} = \llbracket x \rrbracket^{\ell'/2} - \llbracket \Delta \rrbracket^{\ell'/2}$ ;
  - $\llbracket \delta' \rrbracket^{\ell'/2} = \llbracket y \rrbracket^{\ell'/2} - \llbracket \Delta' \rrbracket^{\ell'/2}$ ;
  - reveal  $\delta$  and  $\delta'$  to each other;
- 6:  $P_1$  and  $P_2$  set output as  $\llbracket \mathcal{T}(\delta \cdot 2^{\ell'/2} + \delta') \rrbracket^{\ell'}$ ;

---

**Communication Optimization.** We observe that in cases where two lookup tables share the same input, the communication cost of table evaluation can be further optimized. Specifically, consider lookup table 1 with inputs  $(x, y)$  and lookup table 2 with inputs  $(x, y')$ , where the input  $x$  is reused. Let table 1 use offsets  $\Delta^{(1)}$  and  $\Delta'^{(1)}$ , and table 2 use  $\Delta^{(2)}$  and  $\Delta'^{(2)}$ . By setting  $\Delta^{(1)} = \Delta^{(2)}$ , i.e., using the same offset when generating the two tables, we only need to open  $x - \Delta$  once during the online phase. The revealed value can then be used to query both tables separately. When a large number of lookup tables are involved, this optimization reduces the online communication cost by up to 50%.

**Lookup Table for Share Conversion.** We use lookup tables to perform ring size extension, which expand  $\llbracket x \rrbracket^{\ell'}$  to  $\llbracket x \rrbracket^{\ell}$ , where  $\ell > \ell'$ . That is,  $P_0$  secret shares a lookup table  $\mathcal{T}(i) = i$  over the  $\ell$ -bit ring. Then, using a lookup table evaluation protocol, the input  $\llbracket x \rrbracket^{\ell'}$  is converted into a two-party

secret share  $\llbracket x \rrbracket^\ell$  over the  $\ell$ -bit ring. In addition, we employ lookup tables to convert two-party secret shares into replicated secret shares. Specifically, we convert  $\llbracket x \rrbracket^{\ell'}$  into  $\langle x \rangle^\ell$ . It works as follows. First, we convert  $\llbracket x \rrbracket^{\ell'}$  to  $\llbracket x \rrbracket^\ell$  using a lookup table. After that, a reshare step is performed to obtain RSS  $\langle x \rangle^\ell$ , as follows:

- $P_0$  and  $P_1$  jointly generate  $\langle x \rangle_2^\ell$ ,
- $P_0$  and  $P_2$  jointly generate  $\langle x \rangle_1^\ell$ ,
- Then,  $P_1$  computes  $\delta_1 = \llbracket x \rrbracket_1^\ell - \langle x \rangle_2^\ell$ ,
- $P_2$  computes  $\delta_2 = \llbracket x \rrbracket_2^\ell - \langle x \rangle_1^\ell$ ,
- Finally,  $P_1$  and  $P_2$  open and add the results to obtain  $\langle x \rangle_0^\ell = \delta_1 + \delta_2$ .

We denote the above procedure as  $\langle x \rangle^\ell \leftarrow \Pi_{\text{convert}}^{\ell', \ell}(\llbracket x \rrbracket^{\ell'})$ .

## Linear Layer Evaluation

Linear layer computation mainly includes Fully Connected layers and Matrix Multiplication. We first discuss the case of Fully Connected layers.

**Fully Connection.** The fully connected layer performs matrix operations, which can be reformulated as vector inner products. It takes as input a binary weight vector  $W \in \{-1, 1\}^N$ , a 4-bit activation vector  $x \in \{-8, \dots, 7\}^N$ , and three implicit scaling parameters  $s_w$ ,  $s_x$  and  $s_y$ . The computation process is as follows:

$$y = s_w s_x / s_y \cdot \sum_{i=0}^{N-1} W_i \cdot x_i$$

Analyzing the above formula, since each  $W_i \cdot x_i$  results in a 4-bit number and the sum of  $N$  such 4-bit numbers can be at most a  $4 + \log_2 N$ -bit number, we need to perform the computation of  $\sum W_i \cdot x_i$  over a ring of  $4 + \log_2 N$  bits to prevent overflow. Since using RSS for vector inner product computation incurs very low overhead, we employ RSS to perform these inner product operations. The multiplication by  $\frac{s_w s_x}{s_y}$  poses a significant challenge, as this term is a high-precision value. Executing such a computation under MPC requires fixed-point arithmetic and introduces the need for truncation. Re-examining the calculation, we observe that the purpose of  $\frac{s_w s_x}{s_y}$  is to rescale a  $4 + \log_2 N$ -bit value back down to a 4-bit range. To handle this efficiently, we can scale it by a factor of  $2^{12}$ , which effectively shifts the original lower 4-bit result to the highest 4 bits. At the same time,  $\left\lfloor 2^{12} \cdot \frac{s_w s_x}{s_y} \right\rfloor$  can be treated as an integer, eliminating the need to introduce fixed-point arithmetic. Then we directly extract the top 4 bits as the final 4-bit result, effectively approximating the rescaling via high-bit truncation. Considering that in our BERT model, the dimension of the fully connected layer's inner product is 768, we perform the above computation over a  $4 + 12$ -bit ring, since  $2^{12} > 768$ . The scaling factor is set to  $\left\lfloor 2^{12} \cdot \frac{s_w s_x}{s_y} \right\rfloor$ . Algorithm 3 depicts such procedure. After each multiplication, the output becomes a two-party secret share held by  $P_1$  and  $P_2$ . For each layer which requires 16-bit RSS input, if the output of the previous layer is in the form of 4-bit two-party secret

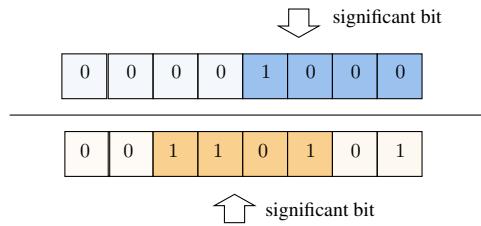


Figure 4: Significant bits of division

sharing, we apply the conversion  $\Pi_{\text{convert}}^{4, 16}$  to restore it to a 16-bit RSS.

**Remark:** We observe that under MPC evaluation, clipping is not necessary. The primary purpose of clipping is to prevent extremely large values from affecting the computation. However, in the MPC setting, we find that values exceeding the threshold are effectively truncated and do not impact the evaluation result.

**Matrix Multiplication.** Quantized matrix multiplication is similar to a fully connected (FC) layer, as it is also implemented through vector dot products. We can adopt a similar evaluation strategy as used in FC layers. Similarly,

Algorithm 3: Inner product for quantized FC.

**Input:**  $\langle \cdot \rangle^{16}$  shares of  $x_i \in [0, 2^3] \cup (2^{16} - 2^3, 2^{16})$ ,  $W'_i = 2^{12} \cdot \frac{s_w s_x}{s_y} \cdot W_i \in [0, 2^{15}] \cup (2^{16} - 2^{15}, 2^{16})$

**Output:** shares of  $\sum W'_i \cdot x_i$  in  $\mathbb{Z}_{2^\ell}$

- 1:  $P_j$  calculates  $y_j = \sum_{i=0}^{N-1} \langle W'_i \rangle_{j-1} \cdot \langle x_i \rangle_{j+1} + \langle W'_i \rangle_{j+1} \cdot \langle x_i \rangle_{j-1} + \langle W'_i \rangle_{j+1} \cdot \langle x_i \rangle_{j+1}$ .
- 2:  $P_0$  sends  $y_0$  to  $P_1$ ;
- 3:  $P_1$  set  $y_1 = \text{trc}(y_0 + y_1, 4)$ ;
- 4:  $P_2$  set  $y_2 = \text{trc}(y_2, 4)$ ;

## Nonlinear Layer Evaluation

Non-linear computation includes Softmax and ReLU, with the main computational overhead coming from the Softmax operation.

**Softmax.** In our framework, the softmax function takes a 4-bit quantized vector as input and outputs a 4-bit quantized softmax result. Similarly, the computation is performed in the dequantized domain. The computation formula is as follows:

$$\text{softmax}(x_i) = \left\lfloor 2^4 \cdot \frac{e^{x_i \cdot s_x}}{\sum_j e^{x_j \cdot s_x}} \right\rfloor$$

Here,  $x_i \in \{0, 1, \dots, 15\}$  is the 4-bit quantized input,  $s_x$  is the input scaling factor used to dequantize the input before exponentiation. The result is rescaled by  $2^4$  to fit back into the 4-bit quantized range.

Our secure evaluation scheme is as follows. First, since computing exponentials of  $e$  can exceed the computation range, we begin by scaling both the numerator and the denominator (Chen et al. 2025; Dong et al. 2023). Given that the maximum value of  $x$  is  $x_o$ , we compute  $e^{s_x(x_i - x_o)}$ ,

	<b>Bits</b>	<b>MNLI-m/mm</b>	<b>QQP</b>	<b>QNLI</b>	<b>SST-2</b>	<b>STS-B</b>	<b>MRPC</b>	<b>RTE</b>	<b>Avg</b>
BERT	32-32	84.9/85.5	91.4	92.1	93.2	90.1	86.3	72.2	87.2
Ours	1-4	82.6/82.9	87.2	90.0	91.3	84.5	85.6	58.8	82.9

Table 1: Accuracy benchmark on GLUE benchmark tasks. **BERT** refers to the original BERT model. **Bits** denotes weight-activation bit-width. Tasks are evaluated on datasets **MNLI**, **QQP**, **QNLI**, **SST-2**, **STS-B**, **MRPC** and **RTE**.

which is always less than or equal to 1. Since the softmax output in the model only requires 4-bit precision, we observe that  $e^{s_x(x_i - x_o)}$  also only needs to be computed with 4-bit precision. It is easy to see that the numerator will always contain at least one value equal to 1, and the denominator will be greater than 1. For any input  $x_j$  such that  $x_j < x_o$ , if  $e^{s_x(x_j - x_o)} < \frac{1}{16}$ , then its contribution to the softmax output will inevitably be zero (after quantization). Therefore, we do not need to retain any data beyond the top 4 bits of  $e^{s_x(x_i - x_o)}$ . In particular, we first employ  $\Pi_{\max}$  to calculate the maximum value  $x_o$ . Then all parties calculate  $x_i - x_o$  locally, and evaluate  $e^{s_x(x_i - x_o)}$  using a lookup table  $\mathcal{T}(x_i - x_o)$ , where input  $x_i - x_o$  to the lookup table. To ensure precision under 4-bit representation, we carefully determine the input bit alignment, as illustrated in Fig. 4. First, the lookup table outputs an 8-bit secret share, where the lower 4 bits contain the valid value and the upper 4 bits are zeros. We then perform addition over the 8-bit ring to prevent overflow. As a result, the denominator may carry over into the higher bits. For the numerator, we extract the lower 4 bits; for the denominator, we extract the middle 4 bits—this is because in BERT, the middle 4 bits are sufficient to preserve the high-order information. Then, we use the previously designed multi-input lookup table protocol  $\Pi_{\text{look}}$  to evaluate the division, where the lookup table is defined as  $\mathcal{T}(x||y) = 2^4 \cdot x/y$ .

**ReLU.** We follow the work of Lu et al. (Lu et al. 2025) and use the lookup table to evaluate the ReLU function. For a 4-bit input, and considering that the ReLU output will be used as input to the FC layer, we let the lookup table directly output 16-bit additive share, which are then reshaped into RSS format.

**LayerNorm.** The layer normalization layer requires the computation of the mean and variance, which are expressed as follows:

$$\text{LN}(x_i) = \gamma \cdot \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

where  $\mu$  is the mean of input, and  $\sigma^2$  is the variance. We perform layer normalization using 4-bit quantized data over a 16-bit ring. Before feeding the data into this layer, we use a lookup table to convert the 4-bit share  $\llbracket x \rrbracket^4$  into 16-bit share  $\llbracket x \rrbracket^{16}$ . Owing to the homomorphic property of mean computation under quantization, the mean can be directly computed from the quantized values. The procedure is as follows:

- For quantized vector  $(\llbracket x_0 \rrbracket^{16}, \dots, \llbracket x_{n-1} \rrbracket^{16})$ , locally calculate  $\llbracket 2^{12}\mu \rrbracket^{16} = \lfloor 2^{12}/n \rfloor \sum_{i=0}^{n-1} \llbracket x_i \rrbracket^{16}$ , which store the result in the highest 4-bit.
- $P_1$  and  $P_2$  set  $\llbracket \mu \rrbracket_i^4 = \text{trc}(\llbracket 2^{12}\mu \rrbracket_i^{16}, 4)$ .

Table 2: Performance comparison with CrypTen and Sigma. “#x” denotes a CPU with x threads.

CrypTen	Sigma	Sigma	Ours				
		GPU	#4	GPU	#4	#20	#96
21551	12311.4	4667.9	1315.4	1165.1	969.5		

Then we convert  $\llbracket \mu \rrbracket^4$  to  $\langle \mu \rangle^{16}$  by invoking  $\Pi_{\text{convert}}^{4,16}$ . Considering that

$$\sigma^2 = \frac{1}{n \cdot s_y} \sum_{i=1}^n (x_i \cdot s_x - \mu \cdot s_x)^2 = \frac{s_x^2}{s_y \cdot n} \sum_{i=1}^n (x_i - \mu)^2$$

where  $s_x$  is the quantized scalar for  $x$ , and  $s_y$  is the quantized scalar for variance, we perform the multiplications over  $\langle \cdot \rangle$  to compute  $\sum_{i=1}^n (x_i - \mu)^2$ . Similar to the mean computation, we multiply the result by  $\lfloor 2^{12} \frac{s_x^2}{s_y \cdot n} \rfloor$  and apply truncation to obtain a 4-bit representation. Consequently, we employ the lookup table with two 4-bit inputs to evaluate the final division.

Table 3: Performance comparison with Lu et al. under the WAN setting.

Sequence Length	Lu. et al	20 threads	96 threads	Speedup
8	8135.61	1239.29	1037.55	$\times 7.84$
16	12143.00	1591.61	1485.85	$\times 8.17$
32	16764.15	2320.04	2143.16	$\times 7.82$

Table 4: Communication cost comparison with CrypTen and Sigma (MB)

Tokens	Our Online	Our Offline	CrypTen	Sigma
8	4.43	29.20	3921	43.28
16	8.87	59.34	8342	89.24
32	17.80	122.46	21114	189.17
64	35.83	260.01	63731	421.09

## Performance Evaluation

In this section, we benchmark our new technique with regard to performance improvement and the effect of accuracy on model inference. We then compare the performance of our quantized BERT model inference with CrypTen (Knott et al. 2022), Sigma (Gupta et al. 2023) and Lu. et al. (Lu

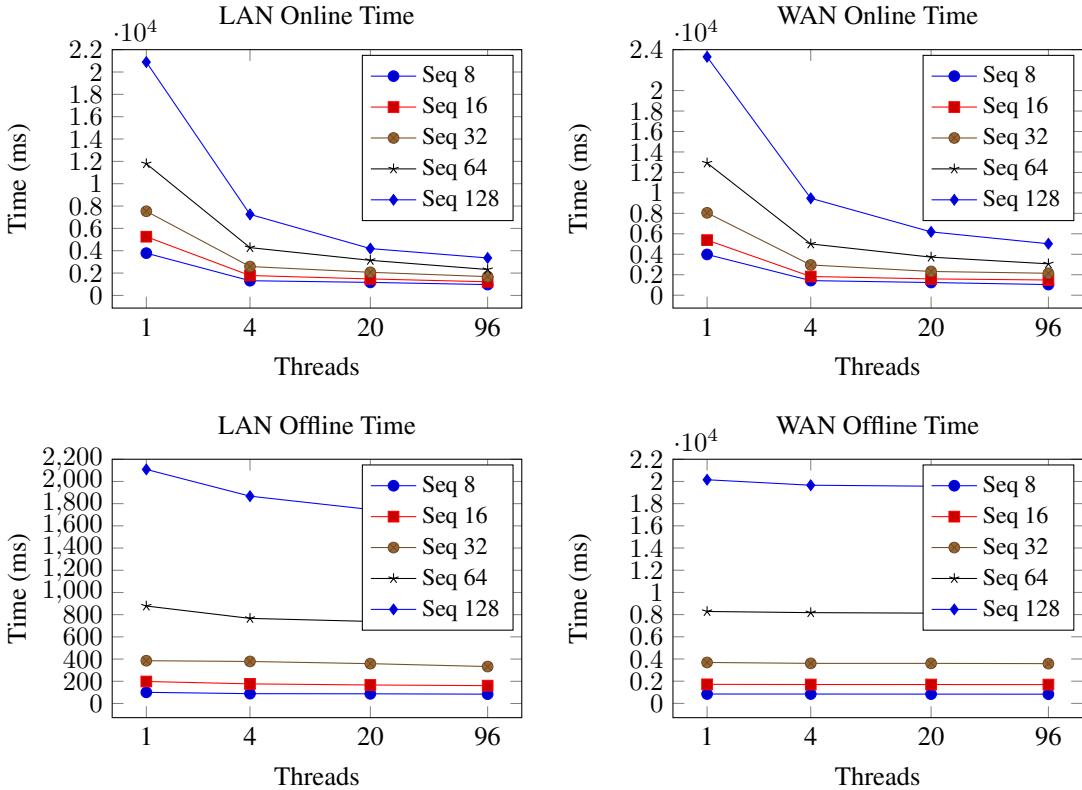


Figure 5: The latency of our privacy-preserving BERT for generating the output of a single token under different network settings, numbers of threads, and input sequence lengths.

et al. 2025). We use three cloud server nodes to simulate three parties, each node with the following configuration: two Intel(R) Xeon(R) E5-2690 v4 @ 2.60GHz CPUs, 64 GiB memory. They are equipped with Ubuntu 16.04.7. We also simulate two different network environments: LAN and WAN corresponded to 5Gbps/100Mbps bandwidth and 0.2ms/40ms round trip latency, respectively. We divide the execution of our framework into two phases: the offline phase and the online phase. Since the random lookup tables can be generated before the inputs are available, we let  $P_0$  generate and distribute these tables during the offline phase.

**Accuracy of Our Scheme** We evaluated the accuracy of our privacy-preserving quantized BERT inference on the GLUE benchmark, as shown in Table 1. ‘BERT’ represents the accuracy of the original full-precision model (32–32 bits), while ‘Ours’ corresponds to our quantized privacy-preserving approach with 1–4 bit precision. Across all GLUE tasks, our method achieves competitive performance with only a moderate drop in average accuracy from 87.2 to 82.9.

**Communication Cost Comparison.** Table. 4 presents the communication cost comparison between our approach and existing privacy-preserving inference frameworks (CrypTen and Sigma) across different sequence lengths. Our method demonstrates significant communication efficiency improvements. Compared to CrypTen, our method achieves a  $885 \sim 1778 \times$  reduction in online communication cost

( $116 \sim 215 \times$  overall when including the offline phase). Against Sigma, our online communication cost shows substantial improvements as  $9.8 \sim 11.8 \times$  reduction in online communication cost and  $1.3 \sim 1.4 \times$  overall cost.

**Performance Benchmark.** Fig. 5 evaluates the latency of our privacy-preserving BERT for generating the output of a single token under different network settings, numbers of threads, and input sequence lengths. The results include both the offline phase and the online phase. It can be observed that increasing the number of threads from 1 to 20 significantly improves the performance of the online phase. With 20 threads and an input length of 8 tokens, the online evaluation of our quantized BERT requires about 1 second. Even with 128 tokens, the evaluation under LAN conditions finishes within 4 seconds.

**Performance Comparison.** We conduct experiments on end-to-end BERT inference, comparing it to CrypTen (Knott et al. 2022) and Sigma (Gupta et al. 2023) on different CPU threads under LAN setting. The results of the experiments are presented in Table. 2. With 4 threads, Sigma takes 12 s, whereas our method only requires 1.3 s, achieving a  $9.36 \times$  speedup over Sigma. We did not implement a GPU-based version; however, even without GPU acceleration, our 96-thread CPU implementation is still  $22 \times$  faster than CrypTen and  $4.8 \times$  faster than Sigma on GPU.

In Table. 3, we compare the performance of our approach with that of Lu et al. under the WAN setting for

quantized privacy-preserving inference. We evaluate input lengths ranging from 8 to 32 tokens, while the results of Lu et al. are reported with 96 threads. With 96 threads, our approach is 7–8× faster than Lu et al., and even with only 20 threads, our method still outperforms their 96-thread implementation by 6.5–7.5×.

## References

- Agrawal, N.; Shahin Shamsabadi, A.; Kusner, M. J.; and Gascón, A. 2019. QUOTIENT: Two-Party Secure Neural Network Training and Prediction. In *CCS*.
- Asharov, G.; Hamada, K.; Ikarashi, D.; Kikuchi, R.; Nof, A.; Pinkas, B.; Takahashi, K.; and Tomida, J. 2022. Efficient Secure Three-Party Sorting with Applications to Data Analysis and Heavy Hitters. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’22, 125–138. New York, NY, USA: Association for Computing Machinery. ISBN 9781450394505.
- Chen, Y.; Tang, Z.; Lu, T.; Zhang, B.; Shi, Z.; and Wang, Z. 2025. Accelerating Private Large Transformers Inference through Fine-grained Collaborative Computation. *IEEE Transactions on Information Forensics and Security*, 1–1.
- Dong, Y.; jie Lu, W.; Zheng, Y.; Wu, H.; Zhao, D.; Tan, J.; Huang, Z.; Hong, C.; Wei, T.; and Chen, W. 2023. PUMA: Secure Inference of LLaMA-7B in Five Minutes. arXiv:2307.12533.
- Gupta, K.; Jawalkar, N.; Mukherjee, A.; Chandran, N.; Gupta, D.; Panwar, A.; and Sharma, R. 2023. SIGMA: Secure GPT Inference with Function Secret Sharing. Cryptology ePrint Archive, Paper 2023/1269.
- Huang, Z.; Lu, W.; Hong, C.; and Ding, J. 2022. Cheetah: Lean and Fast Secure Two-Party Deep Neural Network Inference. In *USENIX Security 2022*, 809–826.
- Jawalkar, N.; Gupta, K.; Basu, A.; Chandran, N.; Gupta, D.; and Sharma, R. 2023. Orca: FSS-based Secure Training with GPUs. Cryptology ePrint Archive, Paper 2023/206.
- Keller, M.; and Sun, K. 2022. Secure Quantized Training for Deep Learning.
- Knott, B.; Venkataraman, S.; Hannun, A.; Sengupta, S.; Ibrahim, M.; and van der Maaten, L. 2022. CrypTen: Secure Multi-Party Computation Meets Machine Learning. arXiv:2109.00984.
- Lang, N.; Sofer, E.; Shaked, T.; and Shlezinger, N. 2023. Joint privacy enhancement and quantization in federated learning. *IEEE Transactions on Signal Processing*, 71: 295–310.
- Liu, Z.; Oğuz, B.; Pappu, A.; Xiao, L.; Yih, S.; Li, M.; Krishnamoorthi, R.; and Mehdad, Y. 2022. BiT: robustly binarized multi-distilled transformer. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS ’22. Red Hook, NY, USA: Curran Associates Inc. ISBN 9781713871088.
- Lu, T.; Zhang, B.; Zhang, X.; and Ren, K. 2025. A New PPML Paradigm for Quantized Models. NDSS.
- Luo, Y.; Xu, N.; Peng, H.; Wang, C.; Duan, S.; Mahmood, K.; Wen, W.; Ding, C.; and Xu, X. 2023. Aq2pnn: Enabling two-party privacy-preserving deep neural network inference with adaptive quantization. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 628–640.
- Mohassel, P.; and Rindal, P. 2018. ABY<sup>3</sup>: A Mixed Protocol Framework for Machine Learning. In *CCS 2018*, 35–52.
- Mohassel, P.; and Zhang, Y. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *S&P 2017*, 19–38.
- Rathee, D.; Rathee, M.; Kumar, N.; Chandran, N.; Gupta, D.; Rastogi, A.; and Sharma, R. 2020. CrypTF2: Practical 2-Party Secure Inference. In *CCS 2020*, 325–342. ACM.
- Riazi, M. S.; Samragh, M.; Chen, H.; Laine, K.; Lauter, K.; and Koushanfar, F. 2019. {XONN}:{XNOR-based} oblivious deep neural network inference. In *USENIX*.
- Wagh, S.; Tople, S.; Benhamouda, F.; Kushilevitz, E.; Mittal, P.; and Rabin, T. 2021. Falcon: Honest-Majority Maliciously Secure Framework for Private Deep Learning. *Proc. Priv. Enhancing Technol.*, 2021(1): 188–208.
- Zhou, L.; Zhang, B.; Wang, Z.; Lu, T.; Song, Q.; Zhang, S.; Cui, H.; and Yu, Y. 2025. On Probabilistic Truncation in Privacy-preserving Machine Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 22955–22964.
- Zhu, H.; Wang, R.; Jin, Y.; Liang, K.; and Ning, J. 2021. Distributed additive encryption and quantization for privacy preserving federated deep learning. *Neurocomputing*, 463: 309–327.
- Zhu, X.; Wang, J.; Chen, W.; and Sato, K. 2023. Model compression and privacy preserving framework for federated learning. *Future Generation Computer Systems*, 140: 376–389.