# A New PPML Paradigm for Quantized Models

Tianpei Lu[1], Bingsheng Zhang[12]✉, Xiaoyuan Zhang[1] and Kui Ren[1]
[1]The State Key Laboratory of Blockchain and Data Security, Zhejiang University,
Email: {lutianpei, bingsheng, zhangxiaoyuan, kuiren}@zju.edu.cn.
[2]Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security

***Abstract*—Model quantization has become a common practice in machine learning (ML) to improve efficiency and reduce computational/communicational overhead. However, adopting quantization in privacy-preserving machine learning (PPML) remains challenging due to the complex internal structure of quantized operators, which leads to inefficient protocols under the existing PPML frameworks.**

**In this work, we propose a new PPML paradigm that is tailor-made for and can benefit from quantized models. Our main observation is that lookup tables can ignore the complex internal constructs of any functions which can be used to simplify the quantized operator evaluation. We view the model inference process as a sequence of quantized operators, and each operator is implemented by a lookup table. We then develop an efficient private lookup table evaluation protocol, and its online communication cost is only $\log n$, where $n$ is the size of the lookup table. On a single CPU core, our protocol can evaluate $2^{26}$ tables with 8-bit input and 8-bit output per second.**

**The resulting PPML framework for quantized models offers extremely fast online performance. The experimental results demonstrate that our quantization strategy achieves substantial speedups over SOTA PPML solutions, improving the online performance by $40 \sim 60\times$ w.r.t. convolutional neural network (CNN) models, such as AlexNet, VGG16, and ResNet18, and by $10 \sim 25\times$ w.r.t. large language models (LLMs), such as GPT-2, GPT-Neo, and Llama2.**

## I. Introduction

Machine Learning (ML) technology has reshaped the way we analyze data, leading to breakthroughs in various sectors such as healthcare, finance, transportation, and science.

At the same time, due to the nature of ML, extensive datasets with sensitive information is collected and processed, raising significant privacy concerns. This has led to an urgent call for the development of Privacy-Preserving Machine Learning (PPML) techniques. Secure Multi-Party Computation (MPC) has emerged as a pivotal cryptographic primitive within the realm of PPML. In a nutshell, MPC allows multiple parties to jointly evaluate a function while keeping their inputs private.

In practice, the communication cost is often the performance bottleneck of an MPC-based PPML. For instance, ResNet-50 [27], with 50 convolution layers and 98MB parameters, requires over 3.8 billion fix-point (or floating-point) MPC operations to complete one model inference task, which produces nearly 2GB of communication, even adopting the most efficient MPC protocols. Therefore, exploring the possibility of reducing the communication cost is the key to speeding up a PPML platform.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.242872
www.ndss-symposium.org

**Quantization.** The quantization technique [71], [16], [15], [32] maps high-precision floating-point values to a smaller set of discrete finite values, and it has been widely adopted to speed up model inference in practice. For large models, quantization is an essential compression technique that could potentially reduce the model size by two to four times without compromising its accuracy. This reduction in data size means that a quantized model requires less memory bandwidth to fetch and store the data, which can be a critical performance bottleneck. Less memory usage also means that more of the model can fit into faster caches, reducing the need to access slower main memory. Therefore, the quantization technique is particularly effective for accelerating model inference using GPUs/NPUs with limited I/O bandwidth.

**Difficulty of adopting quantization to PPML.** Several attempts have been made to adopt the quantization technique to the context of PPML. However, none of the existing solutions are quite successful, and naive adoption cannot save the communication cost in general. The main reason is as follows. Although the (intermediate) data between different model operators is quantized in a more succinct representation format, within each operator, the quantized data should be first recovered to its original high-precision format before the operation, and re-quantize back afterwards. Such a precision recovery step typically requires secure multiplication with the private scaler (with high-precision) as well as a module switch operation from a smaller module, e.g., $2^8$, to a bigger module, e.g., $2^{64}$. Note that the workload of the module switch is usually equivalent to the expensive most significant bit extraction in the MPC setting. For instance, Dalskov *et. al* [17] propose an MPC-based platform that supports quantized model inference, but the resulting PPML scheme needs even more communication than the unquantized version.

As a toy example, suppose we want to perform the convolution operation $\boldsymbol{z} \leftarrow \mathsf{Conv}(\boldsymbol{x}, \boldsymbol{w})$, where $\boldsymbol{x}, \boldsymbol{w}$ and $\boldsymbol{z}$ consist of $\ell$-bit fix-point variables. For $\ell'$-bit quantization, we choose proper $\ell$-bit fix-point scale factor $s_0, s_1, s_2$ and $\ell'$-bit offsets $b_0, b_1, b_2$ such that $\boldsymbol{x} = s_0(\boldsymbol{x'} - b_0)$, $\boldsymbol{w} = s_1(\boldsymbol{w'} - b_1)$, and $\boldsymbol{z} = s_2(\boldsymbol{z'} - b_2)$. The quantized convolution operation $\mathsf{Conv}^*$ takes inputs as $\boldsymbol{x'}$ and $\boldsymbol{y'}$, and it shall output $\boldsymbol{z'} :=$

$\frac{1}{s_2} \cdot \mathsf{Conv}(s_0(\boldsymbol{x'} - b_0), s_1(\boldsymbol{w'} - b_1)) + b_2$. It is easy to see that the operation $\mathsf{Conv}^*$ requires extra steps on top of the original $\mathsf{Conv}$. If $s_i$ are kept in private, $\mathsf{Conv}^*$ usually costs more than the unquantized convolution.

As another line of work, to speed up quantized model inference, Riazi *et. al* [57], Agrawal *et. al* [4] and Keller *et. al* [38] propose to treat the quantization scalers as public variables, and the value of those scalers is limited to the perfect power of 2 to avoid secure multiplication. However, we emphasize that this type of approach might cause severe privacy leakage, also the restriction of the choices of quantization scalers has a negative impact on overall model accuracy.

This prompts our main research question:

> *Does there exist an efficient PPML framework that is tailor-made for and can benefit from quantized models?*

In this work, we answer this question affirmatively by proposing a new PPML paradigm.

**A new paradigm.** As mentioned above, model quantization is particularly effective for operators with limited I/O bandwidth; that is, the input/output of the operators is encoded in some compressed format. We observe that operators with such characteristics can be efficiently evaluated by lookup tables. For operators with $n$-bit input and $m$-bit output, the table size is bounded by $2^n \cdot m$ bits. For common quantized models, $n = 4$-bit or 8-bit. Our work focuses on the two-party (2PC) privacy-preserving model inference setting, where one party, called the server, holds the model in plaintext, and another party, called the client, holds the data. Our paradigm lets the model server prepare the lookup tables for each operator in the offline phase, and then the client privately evaluates those lookup tables in order to obtain the model inference result.

In the literature, several works [33], [26] use lookup tables for the evaluation of non-linear functions, e.g. the activation functions; whereas, in this work, we show that even linear functions can be accelerated by lookup tables in the quantization setting. Take the multiplication operation $y = x \cdot w$ as an example. Let $x = s_0(x' - b_0)$, $w = s_1(w' - b_1)$, and $y = s_2(y' - b_2)$. In our setting, the model holder knows $w'$, $\{s_i\}_{i \in \{0,1,2\}}$, and $\{b_i\}_{i \in \{0,1,2\}}$; therefore, we can re-write the operation as $y' = f(x') := \frac{1}{s_2} \cdot s_1(w' - b_1) \cdot s_0(x' - b_0) - b_2$, where all variables besides $x'$ are hard-coded into the function $f$. Since $x'$ is only 8 bits for 8-bit quantization, the lookup table consists of 256 elements; as we will show later, the online communication cost of this private lookup table evaluation is only 8 bits, which is much less than the cost of the conventional secure (quantized) multiplication.

While applying our technique to PPML, our framework supports lookup-based operator fusion, i.e., the multiple lookup tables can be fused into a single lookup table, and thus the overall cost only equals to single lookup table evaluation.

**Private lookup table evaluation.** There are several works [31], [18], [19], [8] on lookup table evaluation and their usage in the context of PPML. For instance, FLUTE [8] utilizes a boolean circuit to represent the lookup table, and their online communication only depends on the output length of the function; more specifically, if the function output is $\ell$ bits, the online communication of FLUTE is $2\ell$ bits, regardless the input length (or table size). However, those works assume the lookup table is public to everyone, which is not suitable for our case. As mentioned before the model holder will embed the model weights/parameters into the lookup table; therefore, in our work, we study the *private* lookup table evaluation problem, where the lookup table is considered as a private input. Another potential approach for constructing private lookup table evaluation is through private function evaluation (PFE) protocols [35], [63], [68], [25]. However, SOTA incurs significant communication and computational overhead. Our technique is based on the secret shifting protocol proposed by Lu *et.al* [46], where their original protocol is only designed for binary vectors. We extend the protocol to support vectors over a large ring (or field), and apply this shift technique to realize our private lookup evaluation scheme. Compared to the SOTA PFE protocols [35], [63], [68], [25], our protocol is more lightweight and efficient. Its online communication is as low as $\log n$ bits, where $n$ is the table size. On a single CPU core, our protocol can evaluate $10^7$ numbers of lookup tables with 8-bit input and 8-bit output per second (See Appendix. D for related work).

**Performance.** We apply our framework to various machine-learning models. For the convolutional neural network (CNN) models, such as AlexNet, VGG16, and ResNet18, our benchmark shows that our 8-bit quantized PPML framework (single-core CPU only) is $40 \sim 60\times$ faster than the typical 2PC SOTA – Cryptoflow2 [56], and $5 \sim 15\times$ faster than the typical 3PC SOTA – Falcon [65] and Bicoptor [69], even though they use GPU acceleration. For the large language models, such as GPT-2, GPT-Neo, and Llama2, our 8-bit quantized framework (single-core CPU only) achieves $10 \sim 25\times$ performance improvement compared to the SOTA works – CrypTen [40] and Sigma [26].

## II. Preliminaries

**Notation.** The frequently used notations are shown in Table I. We denote $n$-dimension vector as $\boldsymbol{a} := (a_0, \ldots, a_{n-1})$, and $a_i$ be the $i^{\mathsf{th}}$ element of $\boldsymbol{a}$. For notation simplicity, we override the multiplication between a vector and a scalar as $\boldsymbol{a} \cdot b := (a_0 \cdot b, \ldots, a_{n-1} \cdot b)$; similarly, we override the addition between a vector and a scalar as $\boldsymbol{a} + b := (a_0 + b, \ldots, a_{n-1} + b)$. We denote $[n]$ as the index set $\{0, \ldots, n-1\}$. We use letters with the prime to represent the corresponding quantized variables ($x'$ represents the quantized value of $x$). We denote the matrix as the uppercase letter $\mathbf{M} := (m_{i,j})_{i \in [n_1], j \in [n_2]}$ with $n_1 \times n_2$ dimension, and denote the element in the $i^{\mathsf{th}}$ row and $j^{\mathsf{th}}$ column of $\mathbf{M}$ as $m_{(i,j)}$. It can also be represented as $\mathbf{M} := (\boldsymbol{m}_j)_{j \in [n_2]}$ where $\boldsymbol{m}_j$ is $j^{\mathsf{th}}$ column vector, and $\mathbf{M} := (\boldsymbol{m}_i)^T_{i \in [n_2]}$ where $\boldsymbol{m}_i$ is $i^{\mathsf{th}}$ row vector. We define $\mathsf{shift}(\boldsymbol{m}, i)$ as the operation of right circular shifting the vector $\boldsymbol{m}$ by $i$ positions. We use $\mathcal{T}^{\ell_x, \ell_y} := (t_0, \ldots, t_{2^{\ell_x}-1})$ to denote a lookup table with $\ell_x$ bits input and $\ell_y$ bits output. When the semantics are clear, we omit the superscript of $\mathcal{T}$. For the evaluation of $\mathcal{T}$ at position $x$, we represent it as $\mathcal{T}(x)$. We view the lookup table as a vector and denote its $r^{\text{th}}$ item as $t_r \in \mathbb{Z}_{2^{\ell_y}}$. For an operator $\mathsf{op}$, we denote its quantized operator as $\mathsf{op}^*$. Let $(k, n)$-$\mathsf{OT}$ denote the $k$-out-of-$n$ oblivious transfer (OT). We consider 2-out-of-2 secret shares and define the secret share $[\![\cdot]\!]^\ell$ over ring $\mathbb{Z}_{2^\ell}$ as $[\![x]\!]^\ell := ([\![x]\!]_1^\ell \in \mathbb{Z}_{2^\ell}, [\![x]\!]_2^\ell \in \mathbb{Z}_{2^\ell})$ where $x = [\![x]\!]_1^\ell + [\![x]\!]_2^\ell \pmod{2^\ell}$. For simplicity, we use $[\![x]\!]$ when the semantics are clear. We denote the shared vector as $[\![\boldsymbol{x}]\!] := ([\![x_0]\!], \ldots, [\![x_{n-1}]\!])$.

TABLE I: Notations

| Notations | Descriptions |
|---|---|
| $\boldsymbol{a}$ | The vector $\boldsymbol{a} := (a_0, \ldots, a_{n-1})$. |
| $\mathbf{M} := (m_{i,j})_{i\in[n_1],j\in[n_2]}$ | The $n_1 \times n_2$ dimension matrix $M$. |
| $\mathsf{op}^*$ | The quantized operator for $\mathsf{op}$. |
| $[n]$ | The index set $\{0, \ldots, n-1\}$. |
| $[\![x]\!]^\ell := ([\![x]\!]_0^\ell, [\![x[\![_1^\ell)$ | The 2PC secret shares of $x$ over $\mathbb{Z}_{2^\ell}$ where $x = [\![x]\!]_0^\ell + [\![x]\!]_1^\ell \pmod{2^\ell}$. |
| $\mathsf{shift}(\boldsymbol{m}, i)$ | Circular shift the vector $\boldsymbol{m}$ by $i$ positions. |
| $\mathcal{T}^{\ell_x,\ell_y}$ | The lookup table with $\ell_x$-bit input and $\ell_y$-bit output. |
| $\mathcal{T}(x)$ | Evaluate lookup table $\mathcal{T}$ at position $x$. |
| $(k, n)$-$\mathsf{OT}$ | k-out-of-n OT. |

**System Architecture and Threat Model.** As shown in Fig. 1. Our PPML framework can be deployed in outsourcing and client/server (C/S) settings. We assume all the participants are semi-honest where the adversary may attempt to extract private information from her views but she must follow the protocol. In particular, our framework contains four participants, denoted by $\mathcal{P} := \{\mathcal{C}, \mathcal{S}, \mathcal{P}_0, \mathcal{P}_1\}$. Among them, $\mathcal{C}$ is the data client, $\mathcal{S}$ is the model server, $\mathcal{P}_0$ and $\mathcal{P}_1$ are the computing nodes in the outsourcing setting.

In our settings, we assume that the machine learning model is prepared in-prior. In other words, the model inference server can quantize the model parameters in advance and use the quantized models as input in the preprocessing phase. In contrast, the user input will only be determined in the online phase.

Without loss of generality, we define the machine learning model as a sequence of operators: $\mathcal{M} := \{\mathsf{op}^{(0)}, \ldots, \mathsf{op}^{(N-1)}\}$ where $\mathsf{op}$ corresponds to the operator of each layer, and the input vector (for the tensor which is the high-dimension matrix, we convert it to vector) as $\boldsymbol{x}$. In our setting, the operator $\mathsf{op}$ is embedded with all the model weights. For instance, the convolution can be written as $\mathsf{op}^{(i)}(\boldsymbol{x}) := \mathsf{Conv}(\boldsymbol{w}, \boldsymbol{x})$, where $\boldsymbol{w}$ are the corresponding model weights. We denote the evaluation of $\boldsymbol{x}$ in $\mathcal{M}$ as $\boldsymbol{y} = \mathcal{M}(\boldsymbol{x})$.

- In the outsourcing setting, the machine learning model server $\mathcal{S}$ inputs the private model $\mathcal{M}$, and the data client $\mathcal{C}$ inputs the private data $\boldsymbol{x}$. They employ two computing nodes $\{\mathcal{P}_0, \mathcal{P}_1\}$ to perform secure model inference $\mathcal{M}(\boldsymbol{x})$. The computing node $P_j$ for $j \in \{0, 1\}$ holds the secret share $[\![\boldsymbol{x}]\!]_j$ and will not collude with other parties.

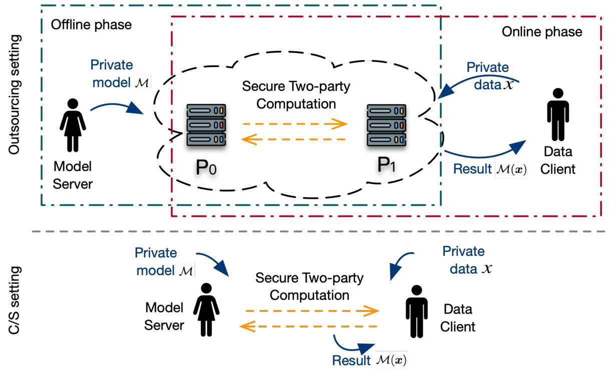

Outsourcing setting
Offline phase
Online phase
Private model $\mathcal{M}$
Secure Two-party Computation
Private data $x$
Model Server
P0
P1
Result $\mathcal{M}(\boldsymbol{x})$
Data Client
C/S setting
Private model $\mathcal{M}$
Private data $x$
Secure Two-party Computation
Model Server
Data Client
Result $\mathcal{M}(\boldsymbol{x})$

Fig. 1: Our system architecture.

- In the C/S setting, instead of employing the computing nodes, the data client $\mathcal{C}$ and the model server $\mathcal{S}$ directly perform secure model evaluation $\mathcal{M}(\boldsymbol{x})$, where $\mathcal{C}$ hold the secret share $[\![\boldsymbol{x}]\!]_0$ and $\mathcal{S}$ hold the secret share $[\![\boldsymbol{x}]\!]_1$.

**Oblivious transfer.** The oblivious transfer is a fundamental cryptographic primitive in which one party (the sender) inputs a list of private messages and another party (the receiver) inputs private indexes. The receiver obtains the messages corresponding to the indexes without any additional information. We denote $(k, n)$-$\mathsf{OT}^\ell$ as the OT with $k$-dimension input indexes and $n$-dimension message list where each message is $\ell$-bit length. We utilize $\Pi_{(1,2)\text{-}\mathsf{OT}^\ell}$ to denote the protocol of $(1, 2)$-$\mathsf{OT}^\ell$ [24], [22]. In $\Pi_{(1,2)\text{-}\mathsf{OT}^\ell}$, the sender inputs message $m_0 \in \mathbb{Z}_{2^\ell}$ and $m_1 \in \mathbb{Z}_{2^\ell}$ into $\Pi_{(1,2)\text{-}\mathsf{OT}^\ell}$; the receiver inputs index $i \in \{0, 1\}$ and receive $m_i$ from $\Pi_{(1,2)\text{-}\mathsf{OT}^\ell}$. Random OT (ROT) [7] is a special case of OT where the selective index is randomly generated by protocol. In an $(n-1, n)$-$\mathsf{ROT}^\ell$, the sends holds a list of messages, and the receiver holds an index $j$ and all the messages except for the $j^{\text{th}}$ message. We utilize $\Pi_{(n-1,n)\text{-}\mathsf{ROT}^\ell}$ [11] to denote the protocol of $(n-1, n)$-$\mathsf{ROT}^\ell$. It sends $(m_0, \ldots, m_n)$ to the sender and sends an index $j \in [n]$ with $n-1$ messages $m_i$ for $i \in [n] \backslash \{j\}$ to the receiver.

**lookup table.** The lookup table $\mathcal{T}$ for operation $\mathsf{op}^* : \{0, 1\}^{\ell_x} \to \{0, 1\}^{\ell_y}$ traverse all possible inputs of $\mathsf{op}^*$. It accepts $\ell_x$ bits input and output $\ell_y$ bits message. The $r^{\text{th}}$ item of lookup table $\mathcal{T}$ stores the result of $\mathsf{op}^*$ with input $r$.

**Secure two-party computation.** Our PPML framework focuses on secure two-party computation (2PC). We define the addition on the secret share as $[\![z]\!] = [\![x]\!] + [\![y]\!]$ and it holds that $z = x + y$ with secret shared form. $P_i$ locally executes $[\![z]\!]_i = [\![x]\!]_i + [\![y]\!]_i$ to obtain the shared result. We use $[\![z]\!] = c \cdot [\![x]\!]$ to denote the scale of a public value, where $z = c \cdot x$. $P_i$ locally executes $[\![z]\!]_i = c \cdot [\![x]\!]_i$ to obtain the shared result. Each computing party can add a private input into the secret share. Specifically, we use $[\![z]\!] = [\![x]\!] + c$ to denote one of parties add private value $c$ to the secret share $[\![x]\!]$, namely, $P_i$ locally sets $[\![z]\!]_i = [\![x]\!]_i + c$ where $P_{1-i}$ sets $[\![z]\!]_i = [\![x]\!]_i$. We define the secret share protocol and the reconstruct protocol as follows.

Protocol $\Pi_{\mathsf{vole}}^{\ell,n}(\boldsymbol{x}, y)$

Input : $\boldsymbol{x} := (x_0, \ldots, x_{n-1}) \in (\mathbb{Z}_{2^\ell})^n$ input by $\mathcal{C}$ and $y \in \mathbb{Z}_{2^\ell}$ input by $\mathcal{S}$.
Output : $\mathcal{C}$ receives $[\![\boldsymbol{z}]\!]_0 := ([\![z_0]\!]_0, \ldots, [\![z_{n-1}]\!]_0) \in (\mathbb{Z}_{2^\ell})^n$, $\mathcal{S}$ receives $[\![\boldsymbol{z}]\!]_1 := ([\![z_0]\!]_1, \ldots, [\![z_{n-1}]\!]_1) \in (\mathbb{Z}_{2^\ell})^n$, where $x_i \cdot y = z_i$ for $i \in [n]$.

**Protocol:**

- $\mathcal{S}$ bit-extract $y$ as $(y_j)_{j\in[\ell]}$ where $y = \sum_{j=0}^{\ell-1} 2^j \cdot y_j$.
- For $j \in [\ell]$, $\mathcal{C}$ and $\mathcal{S}$ invoke $\Pi_{(1,2)\text{-}\mathsf{OT}^{\ell \cdot n}}$:
  - $\mathcal{C}$ picks $\boldsymbol{r}_j := (r_{i,j})_{i\in[n]} \in (\mathbb{Z}_{2^\ell})^n$, inputs $m_0 = -r_{0,j}|| \ldots || - r_{n-1,j}$ and $m_1 = x_0 \cdot 2^j - r_{0,j}|| \ldots ||x_{n-1} \cdot 2^j - r_{n-1,j}$.
  - $\mathcal{S}$ inputs the chooes bit $y_j$ and receives output $r'_j$.
- $\mathcal{C}$ parses $r'_j$ as $r'_j = r'_{0,j}|| \ldots ||r'_{n-1,j}$ and computes $[\![z_i]\!]_0 = \sum_{j=1}^{\ell} r'_{i,j}$, $\mathcal{S}$ sets $[\![z_i]\!]_1 = \sum_{j=0}^{\ell} r_{i,j}$.

Fig. 2: The VOLE protocol based on OT

- $[\![x]\!]^\ell \leftarrow \Pi^\ell_{\mathcal{C}/\mathcal{S}}(\mathbb{P}, x)$: We define the secret share for C/S setting as $\Pi^\ell_{\mathcal{C}/\mathcal{S}}(\mathbb{P}, x)$, where $\mathbb{P} \in \{\mathcal{C}, \mathcal{S}\}$ holds $x$ and secret shares $x$ to $\mathcal{C}$ and $\mathcal{S}$. Before execution, $\mathcal{C}$ and $\mathcal{S}$ generate a correlated seed $\eta$. If $\mathbb{P} = \mathcal{C}$, $\mathcal{C}$ and $\mathcal{S}$ pick $[\![x]\!]^\ell_1 \leftarrow \mathbb{Z}_{2^\ell}$ with the same seed $\eta$. $\mathcal{C}$ locally sets $[\![x]\!]^\ell_0 = x - [\![x]\!]^\ell_1$. Similarly, If $\mathbb{P} = \mathcal{S}$, $\mathcal{C}$ and $\mathcal{S}$ pick $[\![x]\!]^\ell_0 \leftarrow \mathbb{Z}_{2^\ell}$ together. $\mathcal{S}$ locally sets $[\![x]\!]^\ell_1 = x - [\![x]\!]^\ell_0$.
- $[\![x]\!]^\ell \leftarrow \Pi^\ell_{\mathsf{out}}(\mathbb{P}, x)$: We define the secret share for outsourcing setting as $\Pi^\ell_{\mathsf{out}}(\mathbb{P}, x)$, where $\mathbb{P} \in \{\mathcal{C}, \mathcal{S}\}$ holds $x$ and secret shares $x$ to $\mathcal{P}_0$ and $\mathcal{P}_1$. Before execution, $\mathbb{P}$ and $\mathcal{P}_0$ generate a correlated seed $\eta$. $\mathbb{P}$ and $\mathcal{P}_0$ pick $[\![x]\!]^\ell_0 \leftarrow \mathbb{Z}_{2^\ell}$ with same seed $\eta$. $\mathbb{P}$ calculates $[\![x]\!]^\ell_1 = x - [\![x]\!]^\ell_0$ and sends it to $\mathcal{P}_1$.
- $x \leftarrow \Pi^\ell_{\mathsf{rec}\to\mathbf{P}}([\![x]\!]^\ell)$. We define the reconstruction of $[\![x]\!]^\ell$ as $\Pi^\ell_{\mathsf{rec}\to\mathbf{P}}([\![x]\!]^\ell)$. $\mathbf{P}$ is the set of parties to receive $x$. The holders of $[\![x]\!]_0$ and $[\![x]\!]_1$ send them to the parties in $\mathbf{P}$. The parties in $\mathbf{P}$ reconstruct $x = [\![x]\!]_0 + [\![x]\!]_1$.

**Vector Oblivious Linear Evaluation.** Oblivious Linear Evaluation (OLE)[6], [39] is a foundational component in various secure 2PC [59], [55]. In the OLE protocol[6], one party inputs value $x$ and the other inputs value $y$. They jointly evaluate $z = x \cdot y$, and each party obtains the shared result $[\![z]\!]$. Considering the vector case, the primitive so-called Vector Oblivious Linear Evaluation (VOLE) [59], accepts a vector $\boldsymbol{x} := (x_0, \ldots, x_n)$ input by one party and a value $y$ by another party, and output a shared vector of $\boldsymbol{z} := (x_0 \cdot y, \ldots, x_n \cdot y)$. The OLE can be viewed as a special case of VOLE. We follow the OT-based multiplication protocol [37] to realize VOLE. Fig. 2 depicts the procedure of VOLE protocol $\Pi_{\mathsf{vole}}$.

**Fixed-point encoding.** For the floating-point data used in PPML, we first encode it to the fixed-point structure. Specifically, for a fixed point value $x$ with $k$-bit decimal and the effective number of bits $\ell$, if $x \geq 0$, we encode it as $\lfloor x \cdot 2^k \rfloor$; if $x < 0$, we encode it as $2^\ell + \lfloor x \cdot 2^k \rfloor$. For a fixed-point value $x$ with decimal, we utilize encode to denote the procedure of fixed-point encoding, and decode to denote the procedure of decoding to the fixed-point value.

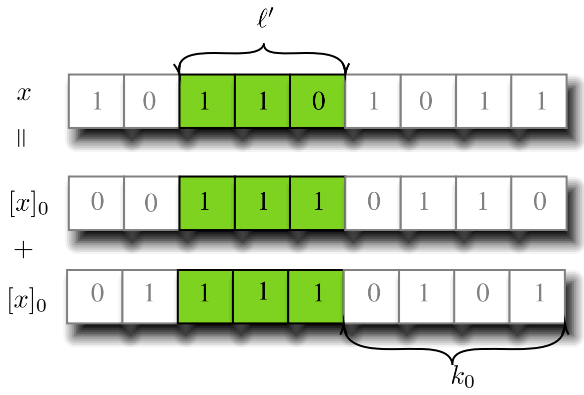


Fig. 3: The case of cut protocol.

**Significant bits extraction.** For a shared value $[\![x]\!]^\ell$, it can be locally extracted of partial significant bits to a lower-precision value [69] with a probabilistic 1-bit carry error. For instance, a fixed-point value with $\ell - \ell'$ decimal bits can drop the decimal bits to obtain an integer. More specifically, we use the function $\mathsf{cut}(x, k, \ell')$ to denote the procedure which drops the first $k$ bits and the last $\ell - \ell' - k$ bits of $x$. Formally, let $x := \sum_{i=0}^{\ell-1} x_i \cdot 2^i$, where $x_i$ is the $i^{\text{th}}$ bit of $x$ from small endian, we have $\mathsf{cut}(x, k, \ell') := \sum_{i=k}^{k+\ell'} x_i \cdot 2^{i-k}$. Based on cut function, we can construct significant bits extraction protocol $[\![x]\!]^{\ell'} \leftarrow \Pi^{\ell'}_{\mathsf{cut}}([\![x]\!]^\ell, k)$ as follows (See Fig 4). $\mathcal{P}_0$ sets $[\![x]\!]^{\ell'}_0 = \mathsf{cut}([\![x]\!]^\ell_0, k, \ell')$ and $\mathcal{P}_1$ sets $[\![x]\!]^{\ell'}_1 = \mathsf{cut}([\![x]\!]^\ell_1, k, \ell')$. Fig. 3 illustrates the case of $\Pi_{\mathsf{cut}}$, the drop of the lower $k$ bits will introduce at most one bit of error (due to two number addition at most case one bit carry), while the drop of higher $\ell - k - \ell'$ bits will not cause any error.

Protocol $\Pi^{\ell'}_{\mathsf{cut}}([\![x]\!]^\ell, k)$

Input : $[\![x]\!]_0 \in \mathbb{Z}_{2^\ell}$ input by $\mathcal{C}$ and $[\![x]\!]_1 \in \mathbb{Z}_{2^\ell}$ input by $\mathcal{S}$.
Output : $\mathcal{C}$ receives $[\![x']\!]^{\ell'}_0 \in \mathbb{Z}_{2^{\ell'}}$, $\mathcal{S}$ receives $[\![x']\!]^{\ell'}_1 \in \mathbb{Z}_{2^{\ell'}}$, where $x' = \mathsf{cut}(x, k, \ell')$.

**Protocol:**

- $\mathcal{C}$ locally calculates $[\![x']\!]^{\ell'}_0 = \mathsf{cut}([\![x]\!]^\ell_0, k, \ell')$.
- $\mathcal{S}$ locally calculates $[\![x']\!]^{\ell'}_1 = \mathsf{cut}([\![x]\!]^\ell_1, k, \ell')$.

Fig. 4: The low precision extraction protocol

## III. Our new framework

In this work, we design a new PPML framework. Our framework achieves performance improvement through two key points. On the one hand, we design our framework tailored for model quantization scenarios. On the other hand, we enable the model server to perform extra operations based on the specific model to enhance the performance of privacy-preserving model evaluation. We address the potential problems of quantized model evaluation in adopting the MPC technique. Consequently, we introduce our PPML paradigm which can evaluate the quantized PPML efficiently.

## *A. Quantization scheme.*

Quantization is the process of mapping continuous infinite values to a smaller set of discrete finite values. In the context of simulation and embedded computing, it is about approximating real-world values with a digital representation that introduces limits on the precision and range of a value.

We formalize the quantization scheme. A quantization scheme is a tuple $\mathcal{Q} := (\mathcal{G}, \mathcal{E}, \mathcal{F}, \mathcal{D})$. Considering the function $f$ to be quantized, the calibration dataset $d$, and $n$-dimension input vector $\boldsymbol{x}$ with $\ell$-bit precision, the quantization scheme $\mathcal{Q}^{d,\ell,\ell'}$ for $\ell'$-bit quantization contains four steps:

- $(\mathcal{F}, \mathcal{E}, \mathcal{D}) \leftarrow \mathcal{G}(f, d)$: the quantization generation $\mathcal{G}$ accept the function $f$, a calibration dataset $d$ and generate a quantized function $f'$, an encode function $\mathcal{E}$, and a decode function $\mathcal{D}$.
- $\boldsymbol{x}' \leftarrow \mathcal{E}(\boldsymbol{x})$: The encode function $\mathcal{E}$ encode the input $\boldsymbol{x} \in \mathbb{Z}^n_{2^\ell}$ to quantized vector $\boldsymbol{x}' \in \mathbb{Z}^n_{2^{\ell'}}$.
- $\boldsymbol{y}' \leftarrow f'(\boldsymbol{x}')$: The quantized function $f'$ is performed on the quantized vector $\boldsymbol{x}'$ and return a quantized output $\boldsymbol{y}'$.
- $\boldsymbol{y} \leftarrow \mathcal{D}(\boldsymbol{y}')$: The decode function $\mathcal{D}$ recover the quantized vector $\boldsymbol{y}'$ to the vector $\boldsymbol{y}$ which is lay on the original precision.

The quantization scheme is designed for IO communication and storage reduction. To measure the magnitude of the reduction, we define the following properties:

**Definition 1** ($\rho$-succinctness)**.** We say the quantization scheme $\mathcal{Q}^{d,\ell,\ell'}$ is $\rho$-succinct, if the precision bits $\ell$ of the original data $x$ and the precision bits $\ell'$ of its quantized data $x'$ hold that:

$$\frac{\ell'}{\ell} = \rho.$$

**Definition 2** ($\varepsilon$-accuracy-loss)**.** We say the quantization scheme $\mathcal{Q}^{d,\ell,\ell'}$ for the calibration dataset $d := \{\boldsymbol{x}_0, \dots, \boldsymbol{x}_{N-1}\}$ is $\varepsilon$-accuracy-loss, if the encode function $\mathcal{E}$, the quantified function $\mathcal{F}$ and the decode function $\mathcal{D}$ hold that:

$$\frac{|\{\mathcal{D}(f'(\mathcal{E}(\boldsymbol{x}_i))) \neq f(\boldsymbol{x}_i); i \in [N]\}|}{N} < \varepsilon;$$

where $|\cdot|$ denotes the number of elements in a collection.

**Quantization in machine learning.** Previous work [71], [16], [15], [32] has demonstrated the effectiveness of quantization techniques in machine-learning scenarios. We give a toy example to illustrate how the quantization scheme works on the machine learning model. Cf. Appendix. E for the quantization examples. We utilize a simple convolutional neural network model $f := (\mathsf{FC}, \mathsf{ReLU}, \mathsf{Conv})$, with input tensor $\boldsymbol{x}$, the weight $\boldsymbol{w}_1$ for the convolution $\mathsf{Conv}$ and $\boldsymbol{w}_2$ for the full connection $\mathsf{FC}$, where $f(\boldsymbol{x}) = \mathsf{FC}(\boldsymbol{w}_2, \mathsf{ReLU}(\mathsf{Conv}(\boldsymbol{w}_1, \boldsymbol{x})))$. We define its quantized model as $f' := (\mathsf{FC}^*, \mathsf{ReLU}^*, \mathsf{Conv}^*)$. We show more details about the quantization scheme $\mathcal{Q}^{d,\ell,\ell'}_{\mathsf{ML}} := (\mathcal{G}, \mathcal{E}, \mathcal{F}, \mathcal{D})$ for the neural network model $f$.

*Encode function* $\mathcal{E}$: Before input $f'$, all the data will be encoded into the quantized data. In the typical ML quantization, the encode function $\mathcal{E}$ is defined as $\boldsymbol{x}' = \mathcal{E}(\boldsymbol{x}) = \lfloor \frac{1}{s} \cdot \boldsymbol{x} \rfloor - b$. Considering the element $x_i$ in the vector $\boldsymbol{x}$ which is a high-precision fixed-point value, it will be scaled by $\frac{1}{s}$ to a low-precision integer. $b$ is an offset which is the so-called zero-point to shift the central value to zero. Note that each input vector $\boldsymbol{x}$ utilizes the single scale factor $s$ and offset $b$ for all elements it contains.

*Quantized function* $f'$: To convert the original model $f$ to a quantized model $f'$, we convert each operator of $f$ to a quantized operator. For the convolution operator $\boldsymbol{y} = \mathsf{Conv}(\boldsymbol{w}_1, \boldsymbol{x})$, the corresponding quantized operator $\mathsf{Conv}^*$ inputs with the quantized vector $\boldsymbol{x}'$ and $\boldsymbol{w}'_1$ and output $\boldsymbol{y}'$. Assume that $s_0$ and $b_0$ are the scale factor of $\boldsymbol{x}$, $s_1$ and $b_1$ are the scale factor of $\boldsymbol{w}_1$, $s_2$ and $b_2$ are the scale factor of $\boldsymbol{y}$. We have $s_2(\boldsymbol{y}' + b_2) = \mathsf{Conv}(s_1(\boldsymbol{w}'_1 + b_1), s_0(\boldsymbol{x}' + b_0))$. From this, the quantized operator for $\mathsf{Conv}$ can be deduced, namely, $\boldsymbol{y}' = \mathsf{Conv}^*(\boldsymbol{w}'_1, \boldsymbol{x}') = \frac{s_1 s_0}{s_2} \cdot \mathsf{Conv}(\boldsymbol{w}'_1 + b_1, \boldsymbol{x}' + b_0) - b_2$. For the next layer, to evaluate $\boldsymbol{y} = \mathsf{ReLU}(\boldsymbol{x})$, the quantized vector holds that $s_2(\boldsymbol{y}' + b_2) = \mathsf{ReLU}(s_1(\boldsymbol{x}' + b_1))$. The quantized operator $\mathsf{ReLU}^*$ can be calculated by $\mathsf{ReLU}^*(x') = \frac{1}{s_2}\mathsf{ReLU}(s_1(\boldsymbol{x}' + b_1)) - b_2$. Similarly, we can infer the quantized operator for full connection $\mathsf{FC}^*$. Consequently, instead of evaluating $f := (\mathsf{Conv}, \mathsf{ReLU}, \mathsf{FC})$ with vector $\boldsymbol{x}$, we evaluate $f' := (\mathsf{Conv}^*, \mathsf{ReLU}^*, \mathsf{FC}^*)$ with the quantized vector $\boldsymbol{x}'$ layer by layer.

*Decode function* $\mathcal{D}$: All the quantized data will be de-quantized (decode to the original precision) before output. It is the inverse operator of the encode function, which is $\boldsymbol{y} = \mathcal{D}(\boldsymbol{y}') = s(\boldsymbol{y}' + b)$. Each integer element $y'_i$ in the vector $\boldsymbol{y}'$ will scale a high-precision fixed-point value $s$ to obtain a high-precision fixed-point value.

*Quantization generation* $\mathcal{G}$: To measure the quantization arguments $\{(s_0, b_0), \dots, (s_m, b_m)\}$, the model server needs to evaluate the model in the calibration data-set $d$. This evaluation is used to generate a priori data ranges for the intermediate results of each operator in the model. These data ranges are then used to determine the scale factor $s$ and the offset $b$. We formalize $\mathcal{G}$ as follows.

- Evaluate $f$ with the calibration data-set $d := (\boldsymbol{x}_0, \boldsymbol{x}_1, \dots, \boldsymbol{x}_{n-1})$. For each operator $\mathsf{op}$, record the maximum and minimum element of output, denoted by $t_{\mathsf{max}}$ and $t_{\mathsf{min}}$.
- For $\mathsf{op}$ with $t_{\mathsf{max}}$ and $t_{\mathsf{min}}$, calculate $s = \frac{t_{\mathsf{max}} - t_{\mathsf{min}}}{2^{\ell'}}$ and $b = \frac{t_{\mathsf{max}} + t_{\mathsf{min}}}{2 \cdot s}$. Calculate $s$ and $b$ for each output wire of the operators.

**The challenge of adopting quantization in PPML.** Although the quantized model $f'$ can significantly reduce the size of input data and the temporary variables, it still can not speed up privacy-preserving machine learning. The main reason is that each quantized operator, e.g., $\mathsf{Conv}^*$, contains a multiplication of $\frac{s_1 s_0}{s_2}$ which lies on the original precision (In practice, equals dequantization and re-quantization). The overall cost of evaluation quantization over typical MPC is even higher than that of the evaluation of the original model. A line of work [57], [4], [38], makes the scale factor public to avoid this problem, which causes massive data leakage on the model.

## *B. Our PPML paradigm*

Revisit the structure of the quantized operator, the input $\boldsymbol{x}'$ and output $\boldsymbol{y}'$ keep a small range (the low precision integer), while $s$ keeps a big range. The quantization scheme is designed to reduce the data communication size of the IO, i.e., between the GPU and memory. With this in mind, we ask whether there exists an MPC operation whose cost depends solely on the input and output sizes, without considering the intermediate computational steps. Interestingly, we observe that if we view the quantized operator as a lookup table, the scale factor $s$ will be hidden within it. For simplicity, we assume the quantized operator $\mathsf{op}^*$ acts independently on each element. Next, we will discuss the operations based on single elements. For vectorized operators, we will address them separately within the context of specific operators. In particular, let the quantized operator $\mathsf{op}^*$ accept input $x' \in \mathbb{Z}_{2^{\ell'}}$ and its hidden parameters $s$ is known to the model server, the model server can locally generate the lookup table as a vector $\mathcal{T} := (\mathsf{op}^*(0), \ldots, \mathsf{op}^*(2^{\ell'}-1))$ by invoking $2^{\ell'}$ times of $\mathsf{op}^*$.

**Quantization scheme in PPML.** We propose the PPML-based quantization scheme $\hat{\mathcal{Q}}_{\mathsf{ML}}^{d,\ell,\ell'}$. The input vector is secret sharing as $[\![x]\!]^{\ell}$, and the quantized function encodes $[\![x]\!]^{\ell}$ to a much succinct secret sharing $[\![x]\!]^{\ell'}$. Considering that PPML involves fixed-point encoding and multiplication will introduce double scaling ($\hat{x} = x \cdot 2^k$, $\hat{y} = y \cdot 2^k$, $\hat{z} = \hat{x} \cdot \hat{y} = x \cdot y \cdot 2^{2k}$ is double scaled), more bits are needed to represent the fixed-point secret sharing in PPML compared to plaintext. For example, 32-bit fixed-point numbers typically use 64-bit encoding to accommodate a single multiplication. Intuitively, for a typical 64-bit PPML, our quantization scheme $\hat{\mathcal{Q}}_{\mathsf{ML}}^{d,\ell,\ell'}$ achieves 0.125-succinctness under 8-bit quantization. Additionally, unlike conventional PPML fixed-point computations that potentially introduce fractional calculation errors, our PPML-based quantization scheme achieves evaluation results identical to plaintext, because the computations are performed on integers, with internal fractional calculations encoded into the lookup table. We test the accuracy loss of the 8-bit quantized convolutional neural networks using the deep learning inference SDK – TensorRT. As shown in Table II, our quantization scheme $\hat{\mathcal{Q}}_{\mathsf{ML}}^{d,\ell,\ell'}$ holds 0.2%-accuracy-loss in 8-bit CNN model.

TABLE II: The accuracy of the 8-bit quantized model compared to the original model.

| | Squeeze Net | ResNet | AlexNet CIFAR | AlexNet Tiny | VGG CIFAR | VGG Tiny |
|---|---|---|---|---|---|---|
| Original Accuracy | 58.19 | 80.24 | 91.52 | 58.63 | 92.72 | 68.08 |
| 8-bit Accuracy | 58.10 | 80.39 | 91.53 | 58.46 | 92.73 | 68.18 |

**Quantized model (lookup table) generation.** The model server $\mathcal{S}$ will first quantize the machine learning model with the calibration data set, obtaining scale factors $(s_i, b_i)$ for each layer. Then the $\mathcal{S}$ generates the lookup table for each quantized operator. We observe that besides the scale factor $s$, $\mathcal{S}$ can also embed the model weight into the lookup table, which makes the dual-input operator, e.g., convolution, matrix multiplication, to single-input. Taking $\mathsf{Mult}^*(w', x') := \frac{1}{s_2}\mathsf{Mult}(s_1(w' + b_1), s_0(x' + b_0)) - b_2$ as an example, the model server knows about $s_1$, $w'$, $b_1$, $s_0$, $b_0$, $s_2$ and $b_2$, such that $w'$ can be part of operator. The operator is converted to a single-input function $\mathsf{Mult}^*(x')$. To generate the lookup table, the model server traverses all possible values of input and evaluates the operator with such values. For $x' \in \{0, \ldots, 2^{\ell'} - 1\}$, the corresponding lookup table is $\mathcal{T}^{\ell',\ell'} := (\mathsf{Mult}^*(w', 0), \ldots, \mathsf{Mult}^*(w', 2^{\ell'} - 1))$. So far, the model server obtains the lookup table for each model operator of $f$. We denote the overall lookup table set for model $f$ which contains $m$ operators as the Q-model $\mathcal{M} := (\mathcal{T}^{(0)}, \ldots, \mathcal{T}^{(m-1)})$, and its $i^{\text{th}}$ lookup table represent the $i^{\text{th}}$ quantized operator.

**Q-model evaluation.** Instead of evaluating the operator with the white box function, the model evaluation turns to the black box with the lookup table. In each layer of the ML model, the model server inputs the private lookup table $\mathcal{T}^{\ell',\ell'} := (\mathsf{op}^*(0), \ldots, \mathsf{op}^*(2^{\ell'}))$, and all parties input the shared value $[\![x']\!]^{\ell'}$ to query $\mathcal{T}$, resulting in a new shared index $[\![y']\!]^{\ell'} = [\![\mathsf{op}^*(x')]\!]^{\ell'} = [\![\mathcal{T}(x')]\!]^{\ell'}$. We directly adopt the lookup evaluation technique on the single-input-single-output (SISO) operator, e.g., activate operator, batch normalization operator, etc. These operators are performed on each element, leading to a single input of the lookup table. For the multiple-input-single-output (MISO) operator, e.g., convolution, and matrix multiplication, we provide the construction in the next section. By querying the lookup table of $\mathcal{M}$ layer by layer, all parties finally obtain the shared output.

**Lookup tables fusion.** We observe that multiple lookup tables can be fused into a single lookup table, resulting in a single fused table rather than multiple table evaluation. The lookup fusion technique is suitable for any SISO operator, or an SISO operator connecting to an MISO operator. Formally, considering a sequence of lookup table $x_2 = \mathcal{T}^{(1)}(x_1), x_3 = \mathcal{T}^{(2)}(x_2), \ldots, x_{n+1} = \mathcal{T}^{(n)}(x_n)$, it holds that $x_{n+1} = \mathcal{T}^{(n)}(\ldots, \mathcal{T}^{(2)}(\mathcal{T}^{(1)}(x_1)), \ldots)$. All the lookup tables can be combined by a single lookup table $x_{n+1} = \mathcal{T}^*(x) := \mathcal{T}^{(n)}(\ldots, \mathcal{T}^{(2)}(\mathcal{T}^{(1)}(x)), \ldots)$. If $\mathcal{T}^{(n)}$ is an MISO lookup table, the fusion concludes an overall MISO lookup table. That is, $\mathcal{T}^*(x, y) := \mathcal{T}^{(n)}(\mathcal{T}^{(1)}(x), \mathcal{T}^{(2)}(y))$. Notice that the fusion of SISO and MISO does not introduce larger lookup tables, while the fusion of MISO and MISO will exponentially increase the size of the lookup table.

**Dequantization fusion.** The other optimization is that we can further reduce the de-quantify phase by fusing the dequantization phase with the previous layer's lookup table. If we view the dequantization scheme as a lookup table, its input range is $\mathbb{Z}_{2^{\ell'}}$ and output range is $\mathbb{Z}_{2^{\ell}}$, such that the lookup table size is $2^{\ell'} \cdot \ell$ (Considering $\ell'$ is small, it is acceptable). Similarly, we use the lookup fusion technique to combine the de-quantify function with the last operator, resulting in a new lookup table with $\mathbb{Z}_{2^{\ell'}}$ input and $\mathbb{Z}_{2^{\ell}}$ output. Note that this technique is not suitable for the quantization operator, as the input of the quantizing function is $\ell$-bit, which would lead to an unacceptably large lookup table of size $2^{\ell} \cdot \ell'$.

## IV. Private lookup Table Evaluation.

### A. The Existing lookup Table Overview

Our PPML framework is based on secure lookup evaluation. Looking forward to a suitable component, we review the existing lookup table evaluation scheme. In general, we classify the lookup table evaluation into two types: i. the scheme where online communication cost only depends on the output size of lookup; ii. the scheme where online communication costs are only dependent on the input size of the lookup.

**Lookup table with Output-length-dependent Cost.** Brüggemann [8] *et. al* propose a lookup table construction named FLUTE where the online phase communication costs are only dependent on the lookup table output size. The FLUTE approach utilizes a boolean circuit to represent the lookup table and adopts the ABY2.0 [53] protocol to evaluate the boolean circuit securely. Since the online phase of ABY2.0 only depends on the output size $\ell'$, the lookup table evaluation only requires concrete $2\ell'$ bits communication. FLUTE provides a fast online phase when the output of the lookup table is small. However, FLUTE is not suitable for our framework, as its lookup table structure needs to be public. For our PPML framework, the lookup table encoding the scale factor $s$, the offset $b$ and the weight $w$ needs to be processed in secret.

**Lookup table with Input-length-dependent Cost.** Another type of lookup evaluation schemes [31], [18], [19] holds the property that the online phase communication costs are only dependent on the lookup table input size, by introducing an offset $r$ on the lookup table. In particular, for a lookup table $\mathcal{T}^{\ell_x,\ell_y}$ whose input size is $2^{\ell_x}$, the preprocessing phase involving two parties jointly generating the shifted shared lookup table $[\![\mathcal{T}']\!]$ whose item is $[\![t_i]\!]^{\ell_y} = [\![\mathcal{T}(i+r)]\!]^{\ell_y}$ for $i \in \{0, \ldots, 2^{\ell_x}-1\}$ ( Circular shifting $\mathcal{T}$ by $r$ position to obtain $\mathcal{T}'$) and the secret shared offset $[\![r]\!]^{\ell_x}$. In the online phase, given the secret shared input $[\![x]\!]$, two parties reconstruct $\delta = x - r$ and set result $[\![y]\!]^{\ell_y} = [\![t_\delta]\!]^{\ell_y}$ (it is easy to see $t_\delta = \mathcal{T}(\delta+r) = \mathcal{T}(x)$). The communication cost of the online phase only contains $2\ell_x$ bits communication of reconstruction which corresponds to the input size. Coincidentally, similar to the output depending on the scheme, these works [31], [18], [19] are also incompatible with our PPML framework, where their settings assume the lookup table is public in the preprocessing phase to generate $[\![t_i]\!]^{\ell_y}$.

**Lookup table with private function evaluation.** The private lookup table evaluation is essential for implementing the quantization scheme. We observe that the private lookup table is a special case of private function evaluation (PFE), and it can be viewed as the distributed oblivious RAM(DORAM), while the dataset is shared, and computing parties pick the selected item using a shared index. Several works [35], [63], [20] realize DORAM based on function secret sharing (FSS), which can also be used to construct the private lookup table evaluation. Specifically, two parties, $P_0$ and $P_1$, hold correlated FSS keys that generate a shared one-hot list. To select an item corresponding to the position of the one-hot from the lookup table, all parties calculate the inner product of the one-hot list and the lookup table. Doerner *et al.* [20] employ an encrypted lookup table where the ciphertext is publicly known to both parties, allowing the inner product computation to remain local in the 2-party computation (2-PC) setting. However, their approach yields the shared query result in ciphertext form, necessitating an additional decryption phase in the MPC setting to obtain the final result. Other attempts [35], [34], [63] avoid using an encrypted lookup table by using replicated secret shared lookup table, but their protocol requires at least three or more computing parties. Moreover, the FSS keys need to evaluate the $O(2^{\ell_x})$ times PRG in the online phase for the lookup table size $2^{\ell_x}$, leading to significant online running time. An alternative approach for realizing the private lookup table evaluation scheme is through the use of garbled circuits. In this scenario, both the lookup table and the index are treated as private inputs. A series of works [29], [28] have explored the private lookup evaluation scheme based on garbled circuits. To the best of our knowledge, David *et al.* [29] achieve SOTA theoretical performance, requiring $(\ell_x - 1)\lambda + \ell_x\ell_y\lambda + 2^{\ell_x}\ell_y$ bits of communication cost, where $\ell_x$ is the input size, $\ell_y$ is the output size, and $\lambda$ is the security parameter (with 256 typically used in garbled circuits) which is still unacceptable for our framework.

### B. Our Private lookup Table Evaluation Scheme.

In this section, we aim to find more efficient and low-communication lookup table evaluation protocols. In our setting, one of the parties inputs the private lookup table $\mathcal{T}^{\ell_x,\ell_y}$, and all parties input a secret shared value $[\![x]\!]^{\ell_x}$. As the lookup table evaluation, all parties receive the shared result $[\![\mathcal{T}(x)]\!]^{\ell_y}$.

Our scheme is based on the aforementioned approach where online communication cost only depends on the input size of the lookup. We observe that this approach is partially compatible with the private lookup table since the shifted lookup table $[\![\mathcal{T}']\!]$ in the online phase is secretly shared and leaks no information about the original lookup table. The remaining challenge is how to generate $[\![\mathcal{T}']\!]^{\ell_y}$ and $[\![r]\!]^{\ell_x}$ with a private lookup table $\mathcal{T}$. Formally, we define the private lookup table evaluation as two phases: (i) private shifted lookup table pair generation, in the preprocessing phase, all parties jointly generate the shifted shared lookup table pair where one of the parties inputs the private lookup table. (ii) lookup table evaluation, in the online phase, we follow the works [31], [18], [19] where all parties are only required to perform $2 \cdot \ell_x$ bits communication to reconstruction. Formally, we formally define the circular shifted lookup pair. We give the definition as follows.

**Definition 3.** Let $\mathcal{T}^{\ell_x,\ell_y}$ be a lookup table with $\ell_x$-bit input and $\ell_y$-bit output. We say a 2PC circular shift lookup pair for

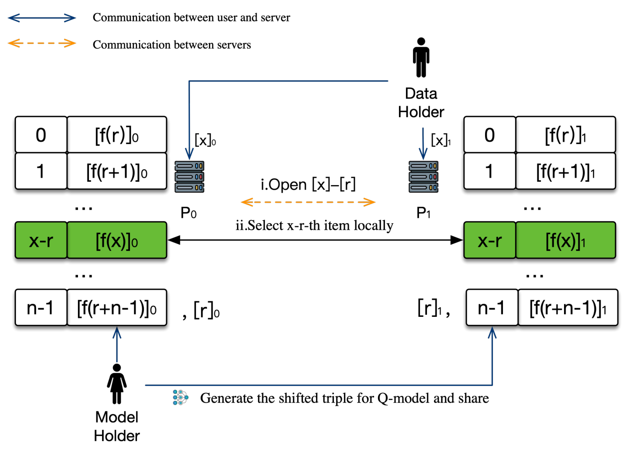


Fig. 5: Secure ML Operator Evaluation with lookup Table.

**Protocol $\Pi^{\ell_x,\ell_y}_{\text{lookup}}(\mathcal{T}^{\ell_x,\ell_y}, [\![x]\!]^{\ell_x})$**

Input : $\mathcal{S}$ inputs a lookup table $\mathcal{T}^{\ell_x,\ell_y} \in (\mathbb{Z}_{2^{\ell_y}})^{2^{\ell_x}}$ in the preprocessing phase. $\mathcal{S}$ and $\mathcal{C}$ inputs a secret shared index $[\![x]\!]^{\ell_x}$ in the online phase.
Output : The secret shared lookup table evaluation result $[\![\mathcal{T}(x)]\!]^{\ell_y}$.

**Preprocessing:**

1) $\mathcal{C}$ and $\mathcal{S}$ invoke $\Pi_{(2^{\ell_x}-1,2^{\ell_x})\text{-ROT}}$:
   - $\mathcal{S}$ receives $(\boldsymbol{m}_i | i \in [2^{\ell_x}], \boldsymbol{m}_i \in (\mathbb{Z}_{2^{\ell_y}})^{2^{\ell_x}})$.
   - $\mathcal{C}$ receives $[\![r]\!]_0$ and $(\boldsymbol{m}_i | i \in [2^{\ell_x}] \backslash \{[\![r]\!]_0\})$.
2) $\mathcal{C}$ picks random $\boldsymbol{m}_{[\![r]\!]_0} \leftarrow (\mathbb{Z}_{2^{\ell_y}})^{2^{\ell_x}}$.
3) $\mathcal{C}$ and $\mathcal{S}$ generate the matrix $\mathbf{M} \in (\mathbb{Z}_{2^{\ell_y}})^{2^{\ell_x} \times 2^{\ell_x}}$ by setting $\boldsymbol{m}_i$ as the $i^{\text{th}}$ column vectors.
4) $\mathcal{C}$ and $\mathcal{S}$ right circular shift the $i^{\text{th}}$ row of $\mathbf{M}$ by $i$ positions locally for $i \in [2^{\ell_x}]$.
5) $\mathcal{S}$ and $\mathcal{C}$ computes $v_i = \sum_{j=0}^{2^{\ell_x}-1} m_{(i,j)}$ and $u_i = \sum_{j=0}^{2^{\ell_x}-1} (-m_{(j,i)})$ for $i \in [2^{\ell_x}]$, and denotes $\boldsymbol{v} := (v_0, \dots, v_{2^{\ell_x}-1})$ and $\boldsymbol{u} := (u_0, \dots, u_{2^{\ell_x}-1})$.
6) $\mathcal{C}$ computes $w_i = v_i + u_{[\![r]\!]_0+i}$, and denotes $\boldsymbol{w} := (w_0, \dots, w_{2^{\ell_x}-1})$.
7) $\mathcal{S}$ picks random $[\![r]\!]_1 \in \mathbb{Z}_{2^{\ell_x}}$ and right circular shift $\mathcal{T}$ by $[\![r]\!]_1$ positions to obtain $\hat{\mathcal{T}}$.
8) $\mathcal{S}$ sends $\boldsymbol{s} = \hat{\mathcal{T}} + \boldsymbol{u}$ to $\mathcal{C}$ and sets $[\![\mathcal{T}']\!]_1 := \boldsymbol{v}$.
9) $\mathcal{C}$ computes $[\![\mathcal{T}']\!]_0 := \text{shift}(\boldsymbol{s}, [\![r]\!]_0) - \boldsymbol{w}$.

**Online:**

1) $\mathcal{S}$ and $\mathcal{C}$ calculates $[\![\delta]\!]^{\ell_x} = [\![x]\!]^{\ell_x} - [\![r]\!]^{\ell_x}$.
2) $\mathcal{C}$ and $\mathcal{S}$ invoke $\delta \leftarrow \Pi^{\ell_x}_{\text{rec}\to\{\mathcal{C},\mathcal{S}\}}([\![\delta]\!]^{\ell_x})$.
3) $\mathcal{C}$ and $\mathcal{S}$ set output $[\![\mathcal{T}(x)]\!]^{\ell_y} = [\![\mathcal{T}'(\delta)]\!]^{\ell_y}$.

Fig. 6: The lookup evaluation protocol $\Pi_{\text{lookup}}$.

lookup table $\mathcal{T}^{\ell_x,\ell_y}$ is $([\![\mathcal{T}']\!]^{\ell_y}, [\![r]\!]^{\ell_x})$, if it holds that $\mathcal{T}'(x) = \mathcal{T}(x+r)$.

**Circular Shift Lookup Pair in Outsourcing Setting.** The circular shift lookup table pair can be generated easily in the outsourcing setting. Since the employed computation parties $\mathcal{P}_0$ and $\mathcal{P}_1$ will not collude with the model holder $\mathcal{S}$, $\mathcal{S}$ can directly generate $r \in \mathbb{Z}_{2^{\ell_x}-1}$ and locally circular shift $\mathcal{T}$ to $r$ position and obtain $\mathcal{T}'$. Consequantly, $\mathcal{S}$ secret share $\mathcal{T}'$ and $r$ to $\mathcal{P}_0$ and $\mathcal{P}_1$.

**Circular Shift Lookup Pair in C/S Setting.** The circular shift lookup pair generation has more challenges in the C/S setting. Recently, Lu [46] *et. al* propose a vector oblivious shift evaluation (VOSE) scheme that can secure shift a $n$-dimension binary vector $\mathcal{T} \in (\mathbb{Z}_2)^n$ to a random position $r$ in the two-party setting. In VOSE, $\mathcal{P}_0$ inputs a binary vector $\mathcal{T} := (t_0, \dots, t_{n-1}) \in (\mathbb{Z}_2)^n$ and receives $[\![\mathcal{T}]\!]^1_0$, $\mathcal{P}_1$ receives $[\![\mathcal{T}]\!]^1_1$ and offset $r$ where $[\![\mathcal{T}]\!]^1_0 \oplus [\![\mathcal{T}]\!]^1_1 = \text{shift}(\mathcal{T}, r)$. We realize that applying their definition to the ring $\mathbb{Z}_{2^{\ell_y}}$ satisfies our requirements. We adopt the technique of Lu *et. al* to our circular shift lookup pair generation. In our setting, we define the VOSE in the arbitrary ring $\mathbb{Z}_{2^\ell}$, namely, $\mathcal{S}$ inputs a vector $\mathcal{T} \in (\mathbb{Z}_{2^\ell})^n$ and receives $[\![\mathcal{T}]\!]_0$, $\mathcal{C}$ receives $[\![\mathcal{T}]\!]_1$ and offset $r$ where $[\![\mathcal{T}]\!]_0 + [\![\mathcal{T}]\!]_1 = \text{shift}(\mathcal{T}, r)$. Using VOSE, we realize the circular shift lookup pair generation. We let $\mathcal{S}$ locally circular shift $\mathcal{T}$ to $r_1$ position and input the shifted lookup table to VOSE. $\mathcal{C}$ input another position $r_0$ to VOSE. Since $\mathcal{S}$ locally shift $\mathcal{T}$ to $r_1$ position and VOSE shift $r_0$ position, the overall shifted position is $r_0 + r_1$. After that $\mathcal{C}$ receive $[\![\mathcal{T}']\!]_0$ and $\mathcal{S}$ receive $[\![\mathcal{T}']\!]_1$ where $\mathcal{T}' = \text{shift}(\mathcal{T}, r_0 + r_1)$. Setting $[\![r]\!]_0 = r_0$ and $[\![r]\!]_1 = r_1$, we obtain the circular shift lookup pair generation.

Our VOSE protocol is also inspired by Lu *et. al* [46], At a high level, the VOSE protocol contains two parts. In the first part, all parties generate a random VOSE, where $\mathcal{T}$ is a random vector, rather than determined by $\mathcal{P}_0$. In the second part, all parties construct VOSE based on random VOSE.

*Random Vector Oblivious Shift Evaluation over Ring.* In our random VOSE scheme, $\mathcal{C}$ receives two $n$-dimension random vectors $\boldsymbol{u} \in (\mathbb{Z}_{2^\ell})^n$ and $\boldsymbol{v} \in (\mathbb{Z}_{2^\ell})^n$; $\mathcal{S}$ receives a offset $r$ and a vector $\boldsymbol{w} \in (\mathbb{Z}_{2^\ell})^n$, and it holds $\boldsymbol{w} = \text{shift}(\boldsymbol{u}, r) + \boldsymbol{v}$. We realize random VOSE from $\ell \cdot n$ length $n-1$ out of $n$ random OT. Specifically, we describe the process as follows.

- $\mathcal{S}$ and $\mathcal{C}$ invoke an instance of $\Pi_{(n-1,n)\text{-ROT}}$. After the protocol, $\mathcal{S}$ receives $n$ messages $(\boldsymbol{m}_0, \dots, \boldsymbol{m}_{N-1})$ and $\boldsymbol{m}_i \in (\mathbb{Z}_{2^\ell})^n$. $\mathcal{C}$ receives $r$ and all messages except for $\boldsymbol{m}_r$. We allow $\mathcal{C}$ pick random $\hat{\boldsymbol{m}}_r \leftarrow (\mathbb{Z}_{2^\ell})^n$. Viewing $(\boldsymbol{m}_0, \dots, \boldsymbol{m}_{n-1})$ as a $n \times n$-dimension matrix $\mathbf{M}$, $\mathcal{S}$ obtains the complete $\mathbf{M}$, while $\mathcal{C}$ can obtain the $\hat{\mathbf{M}}$ with dummy $r^{\text{th}}$ column $\hat{\boldsymbol{m}}_r$.
- $\mathcal{S}$ and $\mathcal{C}$ right circular shift the $i^{\text{th}}$ row of $\mathbf{M}$ ($\hat{\mathbf{M}}$ for $\mathcal{C}$) by $i$ positions for $i \in [n]$, and denote the new matrix as $\mathbf{M}' := (m'_{(i,j)})_{i\in[n],j\in[n]}$ (or $\hat{\mathbf{M}}' := (\hat{m}'_{(i,j)})_{i\in[n],j\in[n]}$).
- $\mathcal{S}$ sets $v_i = \sum_{j=0}^{n-1} m'_{(i,j)}$ and $u_i = \sum_{j=0}^{n-1} (-m'_{(j,i)})$ for $i \in [n]$ to generate $\boldsymbol{v} := (v_0, \dots, v_{n-1})$ and $\boldsymbol{u} := (u_0, \dots, u_{n-1})$. Note that $v_i$ is the sum of $i^{\text{th}}$ row of $\mathbf{M}'$ and $u_i$ is the sum of $i^{\text{th}}$ column of $\mathbf{M}'$.
- Similarly, $\mathcal{C}$ calculates $\hat{v}_i = \sum_{j=0}^{n-1} \hat{m}'_{(i,j)}$ and $\hat{u}_i = \sum_{j=0}^{n-1} (-\hat{m}'_{(j,i)})$ for $i \in [n]$. Let $w_i = \hat{v}_i + \hat{u}_{r+i} \mod 2^\ell$, it holds that $w_i = \hat{v}_i - \hat{m}'_{(i,i+r)} + m'_{(i,i+r)} + \hat{u}_{r+i} + m_{(i,i+r)} - m'_{(i,i+r)} = v_i + u_{r+i}$. Considering $m'_{(i,i+r)}$ is the only item shift by $m'_r$ which correspond to $\hat{m}'_r$ of $\mathcal{C}$,

the calculation of $w_i$ eliminates the same item of $m'_r$ and $\hat{m}'_r$ so that $\mathcal{S}$ can correctly calculate $w_i$. While any other $v_i$ or $u_i$ is random to $\mathcal{S}$. $\mathcal{S}$ set $\boldsymbol{w} := (w_0, \ldots, w_{n-1})$.

Obviously, $\boldsymbol{w}, \boldsymbol{u}$ and $\boldsymbol{v}$ satisfy $\boldsymbol{w} = \mathsf{shift}(\boldsymbol{u}, r) + \boldsymbol{v}$.

*Private lookup Evaluation from VOSE.* Based on the random VOSE, we construct our private lookup evaluation protocol for $\mathcal{T}^{\ell_x,\ell_y}$. Fig. 6 illustrates the specific procedure of lookup evaluation protocol $\Pi_{\mathsf{lookup}}$. Random VOSE output vectors $\boldsymbol{u} \in (\mathbb{Z}_{2^{\ell_y}})^{2^{\ell_x}}$ and $\boldsymbol{v} \in (\mathbb{Z}_{2^{\ell_y}})^{2^{\ell_x}}$ to $\mathcal{S}$, $r_0 \in \mathbb{Z}_{2^{\ell_x}}$ and a vector $\boldsymbol{w} \in (\mathbb{Z}_{2^{\ell_y}})^{2^{\ell_x}}$ to $\mathcal{C}$, such that $\boldsymbol{w} = \mathsf{shift}(\boldsymbol{u}, r_0) + \boldsymbol{v}$. We let $\mathcal{S}$ picks random offset $r_1 \in \mathbb{Z}_{2^{\ell_x}}$ and locally shift $\mathcal{T}$ by $r_1$ position to obtain $\hat{\mathcal{T}}$. Consequently, $\mathcal{S}$ sends $\boldsymbol{s} = \hat{\mathcal{T}} + \boldsymbol{u}$ to $\mathcal{C}$. Then $\mathcal{C}$ sets $[\![\mathcal{T}']\!]_1 = \mathsf{shift}(\boldsymbol{s}, r_0) - \boldsymbol{w}$ and $\mathcal{S}$ sets $[\![\mathcal{T}']\!]_0 = \boldsymbol{v}$. Clearly, $[\![\mathcal{T}']\!]_1 + [\![\mathcal{T}']\!]_0 = \mathsf{shift}(\boldsymbol{s}, r_0) - \mathsf{shift}(\boldsymbol{u}, r_0) = \mathsf{shift}(\hat{\mathcal{T}}, r_0) = \mathsf{shift}(\mathcal{T}, r_1 + r_0)$. For the security proof of our protocol $\Pi_{\mathsf{lookup}}$, we refer the reader to Appendix. A.

## V. PPML for Quantized Models.

In this section, we give a concrete construction for our PPML framework. We first propose the general PPML framework of our outsourcing setting and C/S setting, in which all the operator is viewed as SISO operators. Next, we discuss the special case of the MISO operators.

### *A. The Outsourcing Setting.*

We first talk about how to apply our paradigm to the outsourcing setting. Compared to the C/S structure, the offline of our outsourcing structure is much cheaper. We define the execution procedure of quantization, lookup table evaluation, and de-quantization as follows. Fig. 7 formally illustrates our quantization framework.

- **Input Quantization.** In the outsourcing setting, the data client holds the input vector $\boldsymbol{x} \in (\mathbb{Z}_{2^\ell})^n$ and inputs it in the online phase, while the model server holds the scale factor $s \in \mathbb{Z}_{2^\ell}$, offset $b \in \mathbb{Z}_{2^{\ell'}}$ and input in the preprocessing phase. They would like to employ two computation parties $\mathcal{P}_0$ and $\mathcal{P}_1$ to evaluate $\mathcal{E}(\boldsymbol{x}) := \frac{1}{s} \cdot \boldsymbol{x} - b \in \mathbb{Z}_{2^{\ell'}}$. At the preprocessing phase, we let the model holder $\mathcal{S}$ first encode $\frac{1}{s}$ as the fixed-point encoding, namely $\hat{s} = \mathsf{enc}(\frac{1}{s})$ (Note that enc scale $\frac{1}{s}$ up to $k$ bits), and secret share the VOLE triple $([\![\boldsymbol{r}]\!]^\ell, [\![\hat{s}]\!]^\ell, [\![\boldsymbol{r}\hat{s}]\!]^\ell)$ to $\mathcal{P}_0$ and $\mathcal{P}_1$. In the online phase, the data client encode $\boldsymbol{x}$ as $\hat{\boldsymbol{x}} = \mathsf{enc}(\boldsymbol{x})$ (Similarly, enc scale $\boldsymbol{x}$ up to $k$ bits) and secret share $\hat{\boldsymbol{x}}$ to $\mathcal{P}_0$ and $\mathcal{P}_1$. $\mathcal{P}_0$ and $\mathcal{P}_1$ reveal $\boldsymbol{\delta} = \hat{\boldsymbol{x}} - \boldsymbol{r}$ and calculate $[\![\boldsymbol{x}']\!]^\ell = \boldsymbol{\delta} \cdot [\![\hat{s}]\!]^\ell + [\![\boldsymbol{x}\hat{s}]\!]^\ell$ which is equals to $[\![\hat{\boldsymbol{x}} \cdot \hat{s}]\!]^\ell := (\hat{x_0} \cdot \hat{s}, \ldots, \hat{x_{n-1}} \cdot \hat{s})$. Consequently, we apply $[\![x'_i]\!]^{\ell'} \leftarrow \Pi^{\ell'}_{\mathsf{cut}}([\![\hat{x_i} \cdot \hat{s}]\!]^\ell, 2k)$ for $i \in [n]$ to chop $\ell'$ significant bits (Note that the last $2k$ bits is the fractional part introduced by the multiplication of $\hat{x}$ and $\hat{s}$). For the additive part of $[\![\boldsymbol{x}']\!]^{\ell'} + [\![b]\!]^{\ell'}$, it can be evaluated locally.
- **Q-model Evaluation.** For the lookup table evaluation of each element $x'$ in the vector $\boldsymbol{x}'$, i.e., the lookup table $\mathcal{T}^{\ell',\ell'}$ with input $[\![x']\!]^{\ell'}$ and output $[\![y]\!]^{\ell'} = [\![\mathcal{T}(x')]\!]^{\ell'}$, we adopt the aforementioned shift pair (The procedure is shown in Fig. 5), we let the model server $\mathcal{S}$ shift $\mathcal{T}$ with a random offset $r$, namely, $\mathcal{T}'(i) = \mathcal{T}(i + r)$ for $i \in \mathbb{Z}_{2^{\ell'}}$. In the preprocessing phase, $\mathcal{S}$ secret share $\mathcal{T}'$ and $r$ to $\mathcal{P}_0$ and $\mathcal{P}_1$. In the online phase, $P_j$ for $i \in \{0, 1\}$ calculates $[\![\delta]\!]_j = [\![x']\!]_j - [\![r]\!]_j$ and reconstruct $\Delta$. $P_j$

**Quantization scheme $\Pi^{\mathsf{out}}_{\mathsf{quantize}}(\boldsymbol{x}, \mathcal{E}, f', \mathcal{D})$**

Input : The fractional precision $k$ is common input; the fixed-point encode of input $\hat{\boldsymbol{x}} = \mathsf{encode}(\boldsymbol{x}) \in \mathbb{Z}_{2^\ell}$ is input by $\mathcal{C}$; the scale factor $\hat{s} = \mathsf{encode}(\frac{1}{s}) \in \mathbb{Z}_{2^\ell}, b \in \mathbb{Z}_{2^{\ell'}}$ for encoding function $\mathcal{E}$ is input by $\mathcal{S}$; the quantized lookup table set for model $f' := (\mathsf{op}_0, \ldots, \mathsf{op}_{N-1}, \hat{\mathsf{op}})$ is input by $\mathcal{S}$, where $\hat{\mathsf{op}}$ is the last lookup table which is combined with the dequantization $\mathcal{D}$.
Output : model evaluation result $\boldsymbol{z} := \mathcal{D}(f'(\mathcal{E}(\boldsymbol{x})))$.

**Input Quantization:**

- (Preprocessing) $\mathcal{S}$ picks random vector $\boldsymbol{r} \leftarrow (\mathbb{Z}_{2^\ell})^n$ and invokes $\Pi^\ell_{\mathsf{out}}(\mathcal{S}, \boldsymbol{r})$ ,$\Pi^\ell_{\mathsf{out}}(\mathcal{S}, \hat{s})$ and $\Pi^\ell_{\mathsf{out}}(\mathcal{S}, \boldsymbol{r} \cdot \hat{s})$, where the computing node $P_i$ for $i \in \{0, 1\}$ holds $([\![\boldsymbol{r}]\!]^\ell_i, [\![\hat{s}]\!]^\ell_i, [\![\boldsymbol{r} \cdot \hat{s}]\!]^\ell_i)$;
- $\mathcal{C}$ invokes $\Pi^\ell_{\mathsf{out}}(\mathcal{C}, \hat{\boldsymbol{x}})$ and the computing node $P_i$ for $i \in \{0, 1\}$ holds $[\![\hat{\boldsymbol{x}}]\!]^\ell_i$;
- $P_i$ for $i \in \{0, 1\}$ does:
  - calculate $[\![\boldsymbol{\delta}]\!]^\ell_i = [\![\hat{\boldsymbol{x}}]\!]^\ell_i - [\![\boldsymbol{r}]\!]^\ell_i$ and invoke $\Pi^\ell_{\mathsf{rec}\to\{\mathcal{P}_0,\mathcal{P}_1\}}([\![\boldsymbol{\delta}]\!]^\ell)$ to reconstruct $\boldsymbol{\delta}$.
  - calculate $[\![\boldsymbol{x}']\!]^\ell_i = \boldsymbol{\delta} \cdot [\![\hat{s}]\!]^\ell_i + [\![\boldsymbol{r} \cdot \hat{s}]\!]^\ell_i$.
  - invoke cut function $[\![x'_j]\!]^{\ell'} \leftarrow \Pi^{\ell'}_{\mathsf{cut}}([\![x'_j]\!]^\ell, 2k)$ for $j \in [n], x'_j \in \boldsymbol{x}'$ locally.

**Model evaluation:**

For the operators $(\mathsf{op}_0, \ldots, \mathsf{op}_{N-1})$ with each element $[\![x']\!]^{\ell'}$ of input vector $[\![\boldsymbol{x}']\!]^{\ell'}$ and output $[\![\mathsf{op}^*_j(x')]\!]^{\ell'}$,

- (Preprocessing) $\mathcal{S}$ invokes $\Pi^{\ell'}_{\mathsf{out}}(\mathcal{S}, \mathsf{op}^*(j + r))$ for $j \in \{0, \ldots, 2^{\ell'} - 1\}$ and $\Pi^\ell_{\mathsf{out}}(\mathcal{S}, r)$, where $P_i$ for $i \in \{0, 1\}$ holds the share of lookup table $([\![\mathcal{T}'(0)]\!]_i, \ldots, [\![\mathcal{T}'(2^{\ell'} - 1)]\!]_i)$ and the offset $[\![r]\!]_i$.
- $P_i$ for $i \in \{0, 1\}$does:
  - calculate $[\![\delta]\!]^{\ell'}_i = [\![x']\!]_i - [\![r]\!]_i$ and invoke $\Pi^{\ell'}_{\mathsf{rec}\to\{\mathcal{P}_0,\mathcal{P}_1\}}([\![\delta]\!])$ to reconstruct $\delta$.
  - set $[\![y']\!]^{\ell'}_i = [\![\mathcal{T}'(\delta)]\!]_i$.

**Output Dequantization:**

For the operator $\hat{\mathsf{op}}^*$ which combines the dequantization operator with the last operator and each element $x'$ of input vector $\boldsymbol{x}'$,

- (Preprocessing) $\mathcal{S}$ invokes $[\![\mathcal{T}^{\ell',\ell}(j)]\!] \leftarrow \Pi^\ell_{\mathsf{out}}(\mathcal{S}, \mathsf{op}^*(j + r))$ for $j \in \{0, \ldots, 2^{\ell'} - 1\}$ and $[\![r]\!]^{\ell'} \leftarrow \Pi^{\ell'}_{\mathsf{out}}(\mathcal{S}, r)$.
- $P_i$ for $i \in \{0, 1\}$ does:
  - calculate $[\![\delta]\!]^{\ell'}_i = [\![x']\!]^{\ell'}_i - [\![r]\!]^{\ell'}_i$ and invoke $\Pi^{\ell'}_{\mathsf{rec}\to\{\mathcal{P}_0,\mathcal{P}_1\}}([\![\delta]\!])$ to reconstruct $\delta$.
  - set $[\![\hat{z}]\!]^\ell_i = [\![\mathcal{T}(\delta)]\!]^{\ell'}_i$ and send $[\![\hat{z}]\!]^\ell_i$ to $\mathcal{C}$.
  - $\mathcal{C}$ calculate $z = \mathsf{decode}([\![\hat{z}]\!]^\ell_0 + [\![\hat{z}]\!]^\ell_1)$ and combine all element to vector $\boldsymbol{z}$.

Fig. 7: The quantization PPML scheme in the outsourcing setting

Quantization scheme $\Pi^{\mathcal{C}/\mathcal{S}}_{\mathsf{quantize}}(\boldsymbol{x}, \mathcal{E}, f', \mathcal{D})$

Input : The fractional precision $k$ is common input; $\hat{\boldsymbol{x}} = \mathsf{encode}(\boldsymbol{x}) \in \mathbb{Z}_{2^\ell}$ input by $\mathcal{C}$; $\hat{s} = \mathsf{encode}(\frac{1}{s}) \in \mathbb{Z}_{2^\ell}, b \in \mathbb{Z}_{2^{\ell'}}$ for encoding function $\mathcal{E}$ input by $\mathcal{S}$; $f' := (\mathsf{op}_0, \ldots, \mathsf{op}_{N-1}, \hat{\mathsf{op}})$ input by $\mathcal{S}$, $\hat{\mathsf{op}}$ is combined with the dequantization $\mathcal{D}$.
Output : The model evaluation result $\boldsymbol{z} := \mathcal{D}(f'(\mathcal{E}(\boldsymbol{x})))$.

**Quantize:**

- (Preprocessing) $\mathcal{C}$ picks random $\boldsymbol{r} := (r_0, \ldots, r_{n-1}) \leftarrow (\mathbb{Z}_{2^\ell})^n$;
- (Preprocessing) $\mathcal{S}$ and invokes $[\![\hat{s}]\!]^\ell \leftarrow \Pi_{\mathcal{C}/\mathcal{S}}(\hat{s})$;
- (Preprocessing) $\mathcal{S}$ and $\mathcal{C}$ invokes $[\![\boldsymbol{r}']\!] := ([\![r'_0]\!], \ldots, [\![r'_{n-1}]\!])_{i \in n} \leftarrow \Pi^\ell_{\mathsf{vole}}(\boldsymbol{r}, \hat{s})$;
- $\mathcal{C}$ calculates $\boldsymbol{\delta} = \boldsymbol{x} - \boldsymbol{r} := (\delta_0, \delta_1, \ldots, \delta_{N-1})$ and sends to $\mathcal{S}$.
- $\mathcal{C}$ and $\mathcal{S}$ calculate $[\![x'_i]\!]^\ell = \delta_i \cdot [\![\hat{s}]\!]^\ell + [\![r'_i]\!]^\ell$ for $i \in [n]$;
- $\mathcal{C}$ and $\mathcal{S}$ invoke cut function $[\![x'_i]\!]^{\ell'} \leftarrow \Pi^{\ell'}_{\mathsf{cut}}([\![x'_i]\!]^\ell, 2k)$ for $i \in [n]$ locally and set $[\![\boldsymbol{x}']\!]^{\ell'} = ([\![x'_i]\!]^{\ell'})_{i \in [n]}$.

**Model evaluation/De-quantization:**

- For the quantized operator $\mathsf{op}^*$ with input $[\![\boldsymbol{x}']\!]^{\ell'} := ([\![x'_i]\!]^{\ell'})_{i \in [n]}$ and output $[\![\mathsf{op}^*(x')]\!]^{\ell'}$, $\mathcal{C}$ and $\mathcal{S}$ invoke $[\![y_i]\!]^{\ell'} \leftarrow \Pi^{\ell',\ell'}_{\mathsf{lookup}}((\mathsf{op}^*(0), \ldots, \mathsf{op}^*(2^{\ell'} - 1)), [\![x_i]\!]^{\ell'})$ for $i \in [n]$.
- For the last operator $\hat{\mathsf{op}}^*$ which is combined with the de-quantization, $\mathcal{C}$ and $\mathcal{S}$ invoke $[\![\hat{z}_i]\!]^\ell \leftarrow \Pi^{\ell',\ell}_{\mathsf{lookup}}((\mathsf{op}^*(0), \ldots, \mathsf{op}^*(2^{\ell'} - 1)), [\![x_i]\!]^{\ell'})$ for $i \in [n]$.
- $\mathcal{S}$ sends $[\![\hat{\boldsymbol{z}}]\!]^\ell_1 := ([\![\hat{z}_0]\!]^\ell_1, \ldots, [\![\hat{z}_{n-1}]\!]^\ell_1)$ to $\mathcal{C}$.
- $\mathcal{C}$ reconstruct $\hat{\boldsymbol{z}} = [\![\hat{\boldsymbol{z}}]\!]^\ell_0 + [\![\hat{\boldsymbol{z}}]\!]^\ell_1$ and recover the fixed-point value $\boldsymbol{z} = \mathsf{decode}(\hat{\boldsymbol{z}})$.

Fig. 8: The quantization PPML scheme in C/S setting

then locally sets output as $[\![y]\!]^{\ell'}_j = [\![\mathcal{T}'(\delta)]\!]_j$. Obviously, $\mathcal{T}'(\delta) = \mathcal{T}(\delta + r) = \mathcal{T}(x')$.

- **Output De-quantization.** As mentioned before, the de-quantization function $\hat{\boldsymbol{y}} = \mathcal{E}(\boldsymbol{y}') = s(\boldsymbol{y}' - b)$, where $\boldsymbol{y}' \in (\mathbb{Z}_{2^{\ell'}})^n$ and $\hat{\boldsymbol{y}} \in (\mathbb{Z}_{2^\ell})^n$, can be combined with the previous operator $\hat{\boldsymbol{y}} = s(\mathsf{op}^*(\boldsymbol{y}') - b)$. The model server $\mathcal{S}$ generates the lookup table $\mathcal{T} \in \mathbb{Z}^{2^{\ell'}}_{2^\ell}$ and evaluate it as like the model evaluation. Upon calculating the result $[\![\hat{\boldsymbol{y}}]\!]^\ell$, $\mathcal{P}_0$ and $\mathcal{P}_1$ reconstruct $\hat{\boldsymbol{y}}$ to the data client $\mathcal{C}$. $\mathcal{C}$ invoke decode function $\boldsymbol{y} = \mathsf{decode}(\hat{\boldsymbol{y}})$ to obtain the fixed-point result $\boldsymbol{y}$.

## B. Client/Server structure

Compared to outsourcing setting, the C/S setting has more challenges. In the outsourcing setting, we assume that the server and the computing nodes will not collude, such that the revealed data $\boldsymbol{\delta} = \boldsymbol{x} - \boldsymbol{r}$ will not leak information of $\boldsymbol{x}$. In contrast, in the C/S setting, no matter the multiplication triple $(\boldsymbol{r}, \hat{s}, \boldsymbol{r} \cdot \hat{s})$ or the shift pair $(\mathcal{T}', r)$ can not be directly generated by server $\mathcal{S}$, because the knowledge of $\boldsymbol{r}$ will lead $\mathcal{S}$ learn $\boldsymbol{x}$ from $\boldsymbol{\delta}$. For this concern, we utilize the VOLE protocol $\Pi_{\mathsf{vole}}$ and our 2PC private lookup evaluation protocol $\Pi_{\mathsf{lookup}}$ to realize PPML inference.

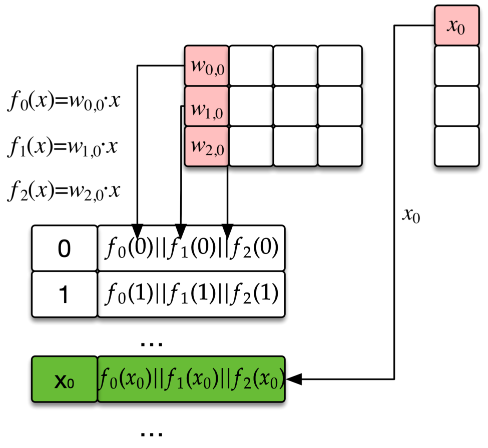


Fig. 9: A illustration example of lookup table evaluation for matrix multiplication.

- **Input Quantization.** In the C/S setting, the data client holds the input vector $\boldsymbol{x} \in (\mathbb{Z}_{2^\ell})^n$ and the model sever holds the scale factor $s \in \mathbb{Z}_{2^\ell}$ and $b \in \mathbb{Z}_{2^{\ell'}}$. They directly invoke a 2PC protocol to evaluate $\mathcal{E}(\boldsymbol{x}) := \frac{1}{s} \cdot \boldsymbol{x} - b \in \mathbb{Z}_{2^{\ell'}}$. Similarly, the model holder encode $\hat{s} = \frac{1}{s}$ at first. At the preprocessing phase, $\mathcal{C}$ generate random shared vector $\boldsymbol{r} := (r_0, \ldots, r_{n-1})$. In particular, $\mathcal{C}$ picks random value $r_i \in \mathbb{Z}_{2^\ell}$ for $i \in [n]$. They adopt $([\![\hat{s} \cdot r_i]\!]^\ell)_{i \in [n]} \leftarrow \Pi_{\mathsf{vole}}(\hat{s}, (r_0, \ldots, r_{n-1}))$, where $\mathcal{C}$ input $(r_i)_{i \in [n]}$ and $\mathcal{S}$ input $\hat{s}$. $\mathcal{S}$ secret share $\hat{s}$ at the same round. In the online phase, $\mathcal{C}$ sends $\boldsymbol{\delta} = \boldsymbol{x} - \boldsymbol{r}$ to $\mathcal{S}$. Similar to the outsourcing setting, all parties calculate $[\![\boldsymbol{x}']\!]^\ell = \boldsymbol{\delta} \cdot [\![\hat{s}]\!]^\ell + [\![\boldsymbol{x}\hat{s}]\!]^\ell$ and perform $\Pi^{\ell'}_{\mathsf{cut}}$ to obtain $[\![\boldsymbol{x}']\!]^{\ell'}$.
- **Q-model Evaluation/Output De-quantization.** We adopt the private lookup table evaluation scheme for the model evaluation in the C/S setting. For each operator $\mathsf{op}^*$ with input $[\![\boldsymbol{x}]\!]^{\ell'}$. $\mathcal{C}$ and $\mathcal{S}$ invoke $[\![y_i]\!]^{\ell'} \leftarrow \Pi^{\ell',\ell'}_{\mathsf{lookup}}((\mathsf{op}^*(0), \ldots, \mathsf{op}^*(2^{\ell'} - 1)), [\![x_i]\!]^{\ell'})$ for each $i \in [n]$. All parties set the output vector $\boldsymbol{y} := (y_0, \ldots, y_{n-1})$. For the operator $\hat{\mathsf{op}}^* : \mathbb{Z}_{2^{\ell'}} \rightarrow \mathbb{Z}_{2^\ell}$ witch contains both model operator and dequantization, $\mathcal{C}$ and $\mathcal{S}$ invoke $[\![\hat{z}_i]\!]^{\ell'} \leftarrow \Pi^{\ell',\ell}_{\mathsf{lookup}}((\hat{\mathsf{op}}^*(0), \ldots, \hat{\mathsf{op}}^*(2^{\ell'} - 1)), [\![x_i]\!]^{\ell'})$. Consequently, $\mathcal{S}$ sends $[\![\hat{z}_i]\!]_1$ to $\mathcal{C}$. $\mathcal{C}$ reconstruct $\hat{\boldsymbol{z}} := (\hat{z}_0, \ldots, \hat{z}_{n-1})$ and invoke decode to recover the fixed-point value.

## C. lookup for MISO.

Above we describe how to evaluate the quantized operator with the lookup table and convert SISO operators of PPML, e.g., activate function, to the lookup table. There remain some operators which are the MISO structure. Since the lookup table size grows exponentially as the input size increases, the lookup scheme is not applicable for the multiple input function. We analyze those MISO operators as follows.

**Multiple lookup table with a single index.** For the convolution layer or the matrix multiplication layer, each private input will multiplicate multiple weights, which can be viewed as a query multiple lookup table with a single index. In particular, as shown in Fig. 9, let $W := (w_{i,j})_{i\in\mathbb{Z}_3, j\in\mathbb{Z}_4}$ be a $3 \times 4$ dimension matrix, and $\boldsymbol{x} := (x_0, \ldots, x_3)$ be a 4 dimension vector. In the calculation of $W \times \boldsymbol{x}$, $x_0$ will multiplicate both $w_{0,0}$, $w_{0,1}$ and $w_{0,2}$. Naively, it can be realized by performing $\Pi_{\mathsf{lookup}}$ three times. Given $f_0(x_0) = w_{0,0} \cdot x_0$, $f_1(x_0) = w_{0,1} \cdot x_0$ and $f_2(x_0) = w_{0,2} \cdot x_0$, we observe that we can combine three functions to an overall function $F(x_0) := f_0(x_0)||f_1(x_0)||f_2(x_0)$. Its corresponding lookup table has $\ell'$ bits input and $3\ell'$ bits output. Considering the online phase of $\Pi_{\mathsf{lookup}}$ only depends on the input size, the online phase cost for $N$ times query lookup table with a single index equals the cost of a single query.

**Addition.** For the addition $y = x_0 + x_1$, where $x_0 = s_0(x'_0 + b_0)$ and $x_1 = s_1(x'_1 + b_1)$ are the temporary variable from two different wires which are unknown to both $\mathcal{C}$ and $\mathcal{S}$, $y = s_2(y' + b_2)$ is the output of addition, its quantized function is $y' = \frac{s_0}{s_2}(x'_0 + b_0) + \frac{s_1}{s_2}(x'_1 + b_1) - b_2$. Addition can be realized using two times invoking lookup table with $f_0(x'_0) := \frac{s_0}{s_2}(x'_0 + b_0)$ and $f_1(x'_1) := \frac{s_1}{s_2}(x'_1 + b_1) - b_2$.

**Convert the dual-input operator to the SISO structure.** For the multiplication $y = x_0 \cdot x_1$, its quantified function is $y' = \frac{s_0 \cdot s_1}{s_2}(x'_0 + b_0)(x'_1 + b_1) - b_2$. A naive idea is to evaluate $x' = (x'_0 + b_0)(x'_1 + b_1)$ using 2PC multiplication in $\mathbb{Z}_{2^{\ell'}}$, after that all parties evaluate $y' = \frac{s_0 \cdot s_1}{s_2} \cdot x' - b_2$ with lookup table. Nevertheless, this approach will incur a significant error caused by overflow of $(x'_0 + b_0)(x'_1 + b_1)$. Considering $(x'_0 + b_0)(x'_1 + b_1) > 2^{\ell'}$, in lookup table, $\frac{s_0 \cdot s_1}{s_2}$ will scale it back to $[0, 2^{\ell'} - 1]$ while separate calculation will lose the overflow part, leading to a big error. The potential solution is to deal $(x'_0 + b_0)(x'_1 + b_1)$ in $\mathbb{Z}_{2^{2\ell'}}$, however, it still enlarged the range of lookup table input. Our solution is to convert the dual-input operator to the SISO structure, and it can be used to deal with arbitrary dual-input operators op$^*$. We set the previous layer of op$^*$ to be $\ell'/2$ output, by encoding the corresponding $\ell'/2$-bit into the previous lookup table. Before inputting two $\ell'/2$-bit values to op$^*$, we combine them to $\ell'$-bit value.

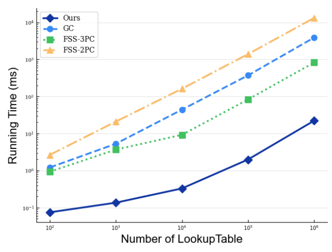
For the other operators' construction for CNN and Transformer, we refer the reader to Appendix. C.

## VI. Implementation and Benchmark.

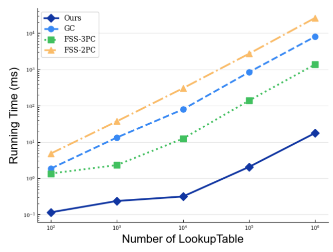
In this section, we benchmark our lookup-based quantization PPML framework. We realize two types of machine learning models – the convolutional neural network for image classification models and the large language models.

### A. *Benchmark setting*

We implement our protocols in C++. For the $\Pi_{\mathsf{OT}}$, we utilize the OT library – libOTe [1]. For our experiments, the CPU version is performed in the server that runs Ubuntu 18.04.2 LTS with Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz, 48 CPUs, 128 GB Memory; the GPU version is performed in the server with the CPU version with extra 4 $\times$ Nvidia RTX 2080ti, 8GB; for the edge devices, we benchmark in the Raspberry Pi 4B with 1.5GHz CPU and 4GB DDR4 Memory. In our benchmark, we set the security parameter $\lambda = 128$. Since most of the code is not available, as a baseline, we use the data in CryptoFlow2 [56], Bicoptor [69], Piranha [67], CrypTen [40], Sigma [26], and use the software to simulate the operating environment in these papers. The benchmarks of our private lookup table evaluation in both outsourcing and C/S settings can be found in Appendix. B.

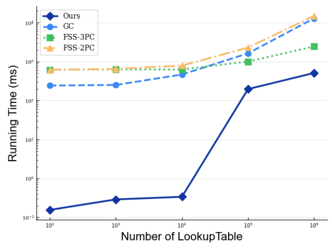


(a) 4-bit Quantization.

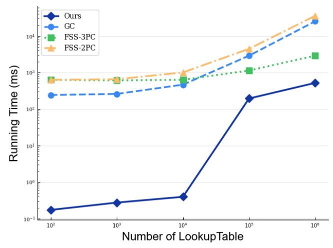


(b) 8-bit Quantization.

Fig. 10: Lookup table evaluation in the LAN setting.



(a) 4bit Quantization.



(b) 8bit Quantization.

Fig. 11: Lookup table evaluation in the WAN setting.

### B. *Private Look-up Table Evaluation.*

We benchmark our private look-up table evaluation against the function secret sharing-based private look-up table eval-

TABLE III: Online performance (for both C/S setting and outsourcing setting) comparison with Piranha-Falcon [67], [65] and Bicoptor [69] for CNN model. (We take $\ell = 64$ and $\ell' = 8$. The bandwidth is 5Gbps/100Mbps in the LAN and the WAN setting respectively and the latency is 0.2ms/40ms respectively.)

| Model | Batch Size | Protocol | LAN | WAN | Plaintext Time |
|---|---|---|---|---|---|
| CIFAR10_ AlexNet | 1650 | P-Falcon | 16.72s | 297.45s | 0.32s |
| | | Bicoptor | 5.00s | 99.83s | |
| | | Ours | 0.98s | 22.86s | |
| Tiny_ AlexNet | 510 | P-Falcon | 30.47s | 513.48s | 0.10s |
| | | Bicoptor | 7.12s | 179.53s | |
| | | Ours | 1.02s | 34.18s | |
| CIFAR10_ VGG16 | 240 | P-Falcon | 54.28s | 968.43s | 0.55s |
| | | Bicoptor | 15.17s | 336.64s | |
| | | Ours | 2.30s | 22.32 | |
| Tiny_ VGG16 | 60 | P-Falcon | 55.02s | 967.74 | 0.15s |
| | | Bicoptor | 15.35s | 336.09s | |
| | | Ours | 3.12s | 20.01s | |

TABLE IV: Performance benchmark of our framework (8bit quantization) on the different device platforms.

| Model | LAN | | | WAN | | |
|---|---|---|---|---|---|---|
| | Raspberry Pi | GPU | CPU(96 Threads) | Raspberry Pi | GPU | CPU(96 Threads) |
| LeNet | 160.09 | 456.13 | 21.33 | 718.34 | 947.76 | 612.49 |
| VGG16-Cirfar10 | 3956.11 | 1530.48 | 44.12 | 4968.66 | 2460.35 | 1413.11 |
| ResNet18-Cirfar10 | 4047.35 | 1714.87 | 44.52 | 5187.35 | 2618.93 | 1488.55 |
| GPT-2(8 input) | 18306.62 | 3261.14 | 42.71 | 25461.80 | 8135.61 | 8384.51 |
| GPT-2(16 input) | 36825.99 | 5953.27 | 54.70 | 44286.03 | 12143.00 | 9141.99 |

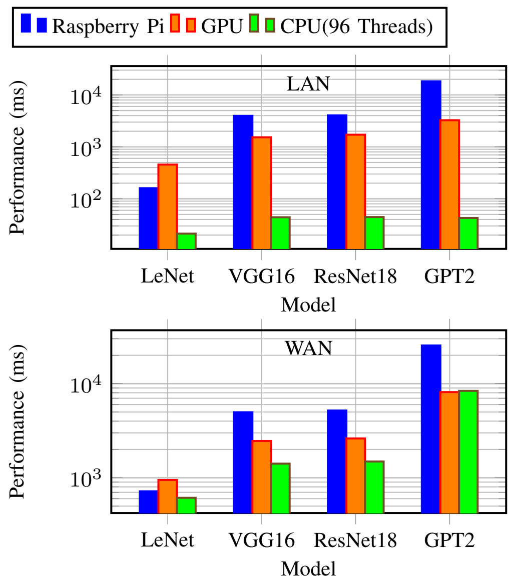


Fig. 12: Performance benchmark of our framework (8bit quantization) on the different device platforms.

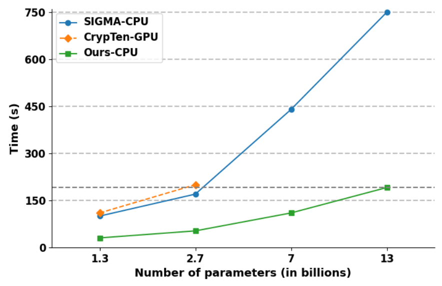
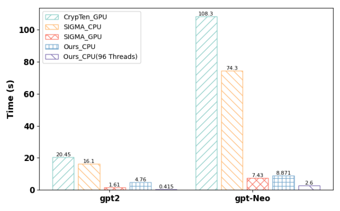
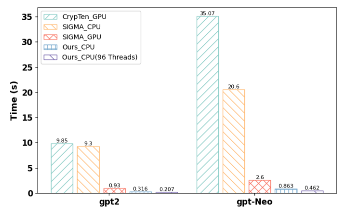


Fig. 13: Online performance comparison with Sigma [26] and CrypTen [40] for LLM (1.3B/2.7B/7B/13B corresponds to GPT-Neo1.3B, GPT-Neo2.7B, Llama2-7B, Llama2-13B respectively) in the LAN setting with 8-bit. The sequence length of the input is 128 and the token output is 1.

uation scheme proposed by Ji [34]. For the 2PC setting of FSS, we apply the technique proposed by Doerner et al [20] and use the garbled circuit to evaluate the decryption circuit of AES (CTR mode). For garbled circuits, we utilize the code from the EMP-tool kit [66] to implement the evaluation. All benchmarks are conducted in the single-threaded. Since David *et al.* [29] do not provide the code, we directly employ



(a) LAN setting



(b) WAN setting

Fig. 14: Online performance comparison for LLM with Sigma [26] and CrypTen [40].(The bandwidth is 9.4 Gbps/293 Mbps in the LAN and the WAN setting respectively and the latency is 0.05 ms / 60 ms respectively. The sequence length of the input is 128 and the output is one token.)

the public lookup table, which compiles the entire lookup table into a circuit and then uses the garbled circuit for evaluation. This approach results in lower communication and computational complexity compared to David et al.'s scheme. Nevertheless, our solution remains two orders of magnitude more efficient than both garbled circuits and FSS in the LAN setting both in 4-bit and 8-bit quantization. Considering the WAN setting, when the data size is sufficiently large, the communication overhead becomes the dominant factor in runtime. In this scenario, our protocol is 5 times more efficient than the FSS protocol and over an order of magnitude more efficient than garbled circuits.

## C. Convolutional neural network evaluation.

For the CNN model, we realize the typical CNN model, like SqueezeNet, ResNet50, AlexNet, and VGG16. We use their 8-bit quantized version in ONNX Model Zoo [2](For SqueezeNet, we manually generate its quantized model). We compare SqueezeNet and ResNet50 with the 2PC framework CryptoFlow2 [56]. We consider the same setting of CryptoFlow2 which performs SqueezeNet and ResNet50 in the ImageNet with the batch size 1 (The dimension of the input image is $224 \times 224 \times 3$). The performance is shown in TABLE. V(Cf. Appendix. B). Our framework achieves more than $40\times$ performance improvement of the online phase compared to CryptoFlow2 in the LAN setting for both SqueezeNet and ResNet50; achieves over $60\times$ performance improvement in the WAN setting. As a trade-off, our protocol introduces a heavy offline phase. Nevertheless, our framework is $2\times$ faster than CryptoFlow2 even adding the offline cost. We compare

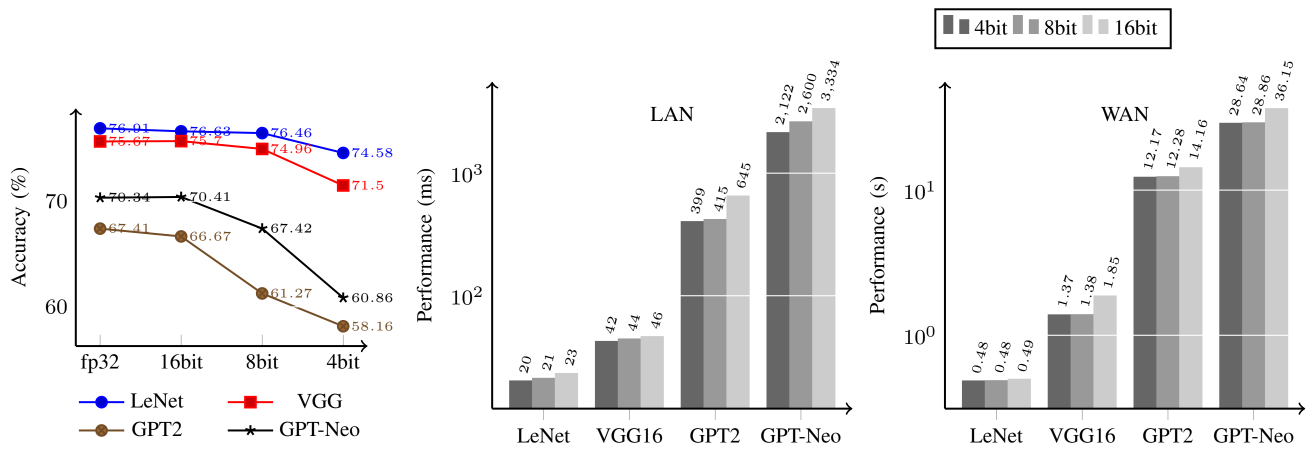


Fig. 15: Performance and Accuracy comparison of 4-bit/8-bit/16-bit quantization in LAN and WAN settings. VGG and LeNet used the CIFAR-10 dataset, while for GPT, we fine-tuned the corresponding text classification model using the MRPC dataset.

the performance of the online phase with the Falcon [65] and Bicoptor [69] for AlexNet and VGG. They are 3PC based on the PPML GPU-platform Piranha [67]. It is worth mentioning that Bicoptor utilizes 8-bit ReLU to realize CNN. We follow their setting where the data set is CIFAR10 ($32 \times 32$ input) and Tiny ImageNet ($64 \times 64 \times 3$ input). Our CPU-based framework realizes more than $5\times$ performance improvement compared to Bicoptor and more than $15\times$ improvement compared to Falcon in both LAN and WAN settings for both AlexNet and VGG16.

## D. LLM model evaluation.

For the LLM model, we realize GPT-2-Base, GPT-Neo1.3B, GPT-Neo2.7B, Llama2-7B and Llama2-13B. The 8-bit quantized model for all of these can be found in Hugging Face [3]. We compare the online phase performance of our framework with Sigma [26] and CrypTen [40]. All of their implementation is based on GPU. Fig. 13 illustrates the performance comparison for GPT-Neo1.3B, GPT-Neo2.7B, Llama2-7B, and Llama2-13B models in the LAN setting. Compared to the CPU version of Sigma, our framework achieves more than $5\times$ performance improvement for all benchmarked LLM models. Even though CrypTen uses GPU acceleration, our framework is over $4\times$ faster than CrypTen for GPT-Neo2.7B. The performance of GPT-2 and GPT-Neo1.3B is depicted in Fig. 14. Our multithreaded version achieves 3 times the performance of the SigmaGPU version on a LAN and 5 times the performance on a WAN. Our framework is $10\times$ faster than the CPU version of Sigma for both GPT-2 and GPT-Neo1.3B in the LAN setting. This improvement will be further amplified over the WAN.

## E. Performance evaluation on different devices.

Fig. 12 and Table. IV depict the performance evaluation across different devices. We benchmark the Raspberry Pi, GPU-equipped server, and the server with 96 threads CPU. Using a multi-threaded CPU yields the best performance. Interestingly, utilizing a GPU does not provide significant performance gains, primarily due to the high time overhead associated with data transfers between the GPU and the host. In our protocol, data frequently moves in and out of the GPU, resulting in memory access overhead dominating the runtime rather than computational overhead. In such scenarios, multi-threading offers better performance improvements. In WAN environments, as the model size increases, network communication becomes the primary performance bottleneck, with the performance of the CPU and GPU becoming comparable. Given the performance limitations of the Raspberry Pi, its model evaluation efficiency is lower than that of the CPU. However, in WAN scenarios, its performance is competitive with that of the CPU, especially for small-scale models. Our results demonstrate that our protocol is highly suitable for multi-threaded devices and exhibits scalability even on resource-constrained devices.

## F. Performance evaluation on quantization bits.

Fig. 15 illustrates the performance and accuracy evaluation of our framework's online phase under different quantization bit settings. The experiments are conducted with 96 threads. For CNN models, 8-bit quantization does not lead to significant accuracy loss (less than 1%), while 4-bit quantization leads to an accuracy drop of under 5%. For transformer models like GPT-2 and GPT-Neo, 16-bit quantization maintains low accuracy loss (less than 1%), while 8-bit quantization results in a more substantial accuracy decrease. In our experiments, the accuracy loss observed in the transformer model is primarily attributed to the layer normalization layer (when we utilize a 16-bit quantized normalization layer, the accuracy loss is close to the overall 16-bit quantization model). In practical applications, it may be advisable to utilize 16-bit quantization or adopt a mixed quantization approach, such as employing 16-bit quantization for the layer normalization layer and 8-bit quantization for other layers. For small-scale models, the choice of quantization bits had minimal impact on the results. However, for transformer models such as GPT, the performance overhead increased with higher quantization bits. In the LAN environment, moving from 4-bit to 8-bit quantization resulted in a 20% increase in runtime, and from 8-bit to 16-

bit, it increased by 25%. In the WAN environment, there was no significant performance difference between 4-bit and 8-bit quantization, but the time overhead increased by 30% when moving from 8-bit to 16-bit. Our results suggest that 8-bit quantization is the optimal choice for balancing efficiency and accuracy in our framework.

## Acknowledgement

This work was supported in part by the National Natural Science Foundation of China under Grant 62072401 and Grant 62232002, in part by Input Output (iohk.io), in part by Ant Group, and in part by Cybersecurity College Student Innovation Funding. (Corresponding author: Bingsheng Zhang.)

## References

[1] libote. https://github.com/osu-crypto/libOTe.

[2] Onnx model zoo. https://github.com/onnx/models.

[3] Quantized gpt-2 model. https://huggingface.co/ybelkada/gpt2-xl-8bit.

[4] Nitin Agrawal, Ali Shahin Shamsabadi, Matt J. Kusner, and Adrià Gascón. Quotient: Two-party secure neural network training and prediction. In *CCS*, 2019.

[5] Yoshimasa Akimoto, Kazuto Fukuchi, Youhei Akimoto, and Jun Sakuma. Privformer: Privacy-preserving transformer with mpc. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 392–410, 2023.

[6] Carsten Baum, Daniel Escudero, Alberto Pedrouzo-Ulloa, Peter Scholl, and Juan Ramón Troncoso-Pastoriza. Efficient protocols for oblivious linear function evaluation from ring-lwe. *Journal of Computer Security*, 30(1):39–78, 2022.

[7] Donald Beaver. Precomputing oblivious transfer. In *Annual International Cryptology Conference*, pages 97–109. Springer, 1995.

[8] A. Brüggemann, R. Hundt, T. Schneider, A. Suresh, and H. Yalame. Flute: Fast and secure lookup table evaluations. In *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 515–533. IEEE Computer Society, 2023.

[9] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. FLASH: fast and robust framework for privacy-preserving machine learning. *Proc. Priv. Enhancing Technol.*, 2020(2):459–480, 2020.

[10] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[11] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. Secret-shared shuffle. In *Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part III 26*, pages 342–372. Springer, 2020.

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

[13] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4pc framework for privacy preserving machine learning. In *NDSS 2020*, 2020.

[14] Tianyu Chen, Hangbo Bao, Shaohan Huang, Li Dong, Binxing Jiao, Daxin Jiang, Haoyi Zhou, Jianxin Li, and Furu Wei. The-x: Privacy-preserving transformer inference with homomorphic encryption. In *ACL*, 2022.

[15] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv preprint*, 2017.

[16] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. Model compression and acceleration for deep neural networks: The principles, progress, and challenges. *IEEE Signal Processing Magazine*, 2018.

[17] Anders Dalskov, Daniel Escudero, and Marcel Keller. Secure evaluation of quantized neural networks, 2020.

[18] Ivan Damgård and Rasmus Zakarias. Fast oblivious aes a dedicated application of the minimac protocol. In David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT*, 2016.

[19] Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni, and Michael Zohner. Pushing the communication barrier in secure computation using lookup tables. NDSS, 2018.

[20] Jack Doerner and Abhi Shelat. Scaling oram for secure computation. In *CCS*, 2017.

[21] Ye Dong, Wen jie Lu, Yancheng Zheng, Haoqi Wu, Derun Zhao, Jin Tan, Zhicong Huang, Cheng Hong, Tao Wei, and Wenguang Chen. Puma: Secure inference of llama-7b in five minutes, 2023.

[22] Nico Döttling, Sanjam Garg, Mohammad Hajiabadi, Daniel Masny, and Daniel Wichs. Two-round oblivious transfer from cdh or lpn. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 768–797. Springer, 2020.

[23] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for mpc over mixed arithmetic-binary circuits. In *CRYPTO*, 2020.

[24] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28(6):637–647, 1985.

[25] Daniel Günther, Ágnes Kiss, and Thomas Schneider. More efficient universal circuit constructions. In *ASIACRYPT*, 2017.

[26] Kanav Gupta, Neha Jawalkar, Ananta Mukherjee, Nishanth Chandran, Divya Gupta, Ashish Panwar, and Rahul Sharma. SIGMA: Secure GPT inference with function secret sharing. Cryptology ePrint Archive, Paper 2023/1269, 2023. https://eprint.iacr.org/2023/1269.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[28] David Heath and Vladimir Kolesnikov. One hot garbling. In *CCS*, 2021.

[29] David Heath, Vladimir Kolesnikov, and Lucien K. L. Ng. Garbled circuit lookup tables with logarithmic number of ciphertexts. In *EUROCRYPT 2024*, 2024.

[30] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. Cheetah: Lean and fast secure {Two-Party} deep neural network inference. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 809–826, 2022.

[31] Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. On the power of correlated randomness in secure computation. In *Theory of Cryptography*, 2013.

[32] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *CVPR*, 2018.

[33] Neha Jawalkar, Kanav Gupta, Arkaprava Basu, Nishanth Chandran, Divya Gupta, and Rahul Sharma. Orca: FSS-based secure training and inference with GPUs. In *S&P*, 2024.

[34] Keyu Ji, Bingsheng Zhang, Tianpei Lu, Lichun Li, and Kui Ren. Uc secure private branching program and decision tree evaluation. *TDSC*, 2023.

[35] Keyu Ji, Bingsheng Zhang, Tianpei Lu, and Kui Ren. Multi-party private function evaluation for RAM. TIFS, 2022.

[36] Wen jie Lu, Zhicong Huang, Zhen Gu, Jingyu Li, Jian Liu, Cheng Hong, Kui Ren, Tao Wei, and WenGuang Chen. BumbleBee: Secure two-party inference framework for large transformers. Cryptology ePrint Archive, Paper 2023/1678, 2023.

[37] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[38] Marcel Keller and Ke Sun. Secure quantized training for deep learning, 2022.

[39] Florian Kerschbaum, Erik-Oliver Blass, and Rasoul Akhavan Mahdavi. Faster secure comparisons with offline phase for efficient private set intersection. In *NDSS*, 2023.

[40] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. Crypten: Secure multi-party computation meets machine learning, 2022.

[41] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow: Secure tensorflow inference. In *S&P 2020*, pages 336–353, 2020.

[42] Sudhakar Kumawat and Hajime Nagahara. Privacy-preserving action recognition via motion difference quantization. In *European Conference on Computer Vision*, pages 518–534. Springer Nature Switzerland, Cham, 2022.

[43] Natalie Lang, Elad Sofer, Tomer Shaked, and Nir Shlezinger. Joint privacy enhancement and quantization in federated learning. *IEEE Transactions on Signal Processing*, 71:295–310, 2023.
[44] Dacheng Li, Rulin Shao, Hongyi Wang, Han Guo, Eric P. Xing, and Hao Zhang. Mpcformer: fast, performant and private transformer inference with mpc, 2023.
[45] Minghui Li, Sherman S. M. Chow, Shengshan Hu, Yuejing Yan, Chao Shen, and Qian Wang. Optimizing privacy-preserving outsourced convolutional neural network predictions. *IEEE Trans. Dependable Secur. Comput.*, 19(3):1592–1604, 2022.
[46] Tianpei Lu, Xin Kang, Bingsheng Zhang, Zhuo Ma, Xiaoyuan Zhang, Yang Liu, and Kui Ren. Efficient 2PC for constant round secure equality testing and comparison. Cryptology ePrint Archive, Paper 2024/949, 2024.
[47] Yukui Luo, Nuo Xu, Hongwu Peng, Chenghong Wang, Shijin Duan, Kaleel Mahmood, Wujie Wen, Caiwen Ding, and Xiaolin Xu. 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*, pages 628–640, 2023.
[48] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *USENIX Security 2020*, pages 2505–2522, 2020.
[49] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 35–52, New York, NY, USA, 2018. Association for Computing Machinery.
[50] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *S&P 2017*, pages 19–38, 2017.
[51] Arpita Patra Ajith Suresh Nishat Koti, Mahak Pancholi. Swift: Super-fast and robust privacy-preserving machine learning. In *USENIX*, 2021.
[52] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. BOLT: Privacy-preserving, accurate and efficient inference for transformers. Cryptology ePrint Archive, Paper 2023/1893, 2023.
[53] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. Aby2. 0: Improved {Mixed-Protocol} secure {Two-Party} computation. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2165–2182, 2021.
[54] Arpita Patra and Ajith Suresh. BLAZE: blazing fast privacy-preserving machine learning. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.
[55] Srinivasan Raghuraman and Peter Rindal. Blazing fast psi from improved okvs and subfield vole. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2505–2517, New York, NY, USA, 2022. Association for Computing Machinery.
[56] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow2: Practical 2-party secure inference. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 325–342, 2020.
[57] M Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin Lauter, and Farinaz Koushanfar. {XONN}:{XNOR-based} oblivious deep neural network inference. In *USENIX*, 2019.
[58] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *AsiaCCS 2018*, pages 707–721, 2018.
[59] Peter Rindal and Phillipp Schoppmann. Vole-psi: Fast oprf and circuit-psi from vector-ole. In *Advances in Cryptology – EUROCRYPT 2021*, pages 901–930, Cham, 2021. Springer International Publishing.
[60] Jinhyun So, Başak Güler, and A. Salman Avestimehr. Codedprivateml: A fast and privacy-preserving framework for distributed machine learning. *IEEE Journal on Selected Areas in Information Theory*, 2(1):441–451, 2021.
[61] Kyle Storrier, Adithya Vadapalli, Allan Lyons, and Ryan Henry. Grotto: Screaming fast $(2+1)$-PC for $\mathbb{Z}_{2^n}$ via (2, 2)-DPFs. Cryptology ePrint Archive, Paper 2023/108, 2023. https://eprint.iacr.org/2023/108.
[62] Sijun Tan, Brian Knott, Yuan Tian, and David J. Wu. Cryptgpu: Fast privacy-preserving machine learning on the gpu. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1021–1038. IEEE, 2021.
[63] Adithya Vadapalli, Ryan Henry, and Ian Goldberg. Duoram: A bandwidth-efficient distributed ORAM for 2- and 3-party computation. USENIX Security, 2023.
[64] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. *Proc. Priv. Enhancing Technol.*, 2019(3):26–49, 2019.
[65] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *Proc. Priv. Enhancing Technol.*, 2021(1):188–208, 2021.
[66] Xiao Wang, Alex J Malozemoff, and Jonathan Katz. Emp-toolkit: Efficient multiparty computation toolkit, 2016.
[67] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. Piranha: A gpu platform for secure computation, 2022.
[68] Shuoyao Zhao, Yu Yu, Jiang Zhang, and Hanlin Liu. Valiant's universal circuits revisited: an overall improvement and a lower bound. ASIACRYPT, 2018.
[69] Lijing Zhou, Qingrui Song, Su Zhang, Ziyu Wang, Xianggui Wang, and Yong Li. Bicoptor 2.0: Addressing challenges in probabilistic truncation for enhanced privacy-preserving machine learning, 2024.
[70] Hangyu Zhu, Rui Wang, Yaochu Jin, Kaitai Liang, and Jianting Ning. Distributed additive encryption and quantization for privacy preserving federated deep learning. *Neurocomputing*, 463:309–327, 2021.
[71] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint*, 2017.
[72] Xi Zhu, Junbo Wang, Wuhui Chen, and Kento Sato. Model compression and privacy preserving framework for federated learning. *Future Generation Computer Systems*, 140:376–389, 2023.

## Appendix

### A. Security of Private lookup Table Evaluation

**Universal Composability (UC).** Our protocols ensure security within the standard semi-honest setting. In this scenario, the adversary may attempt to extract private information from legitimate messages but must adhere strictly to the protocol's procedure. The security proof is based on the Universal Composability (UC) framework [10], which follows the simulation-based security paradigm. In the UC framework, protocols are executed across multiple interconnected machines. The network adversary Adv is allowed to partially control the communication tapes of all uncorrupted machines, observing messages sent to/from uncorrupted parties and influencing message sequences. Then, a protocol $\Pi$ is considered UC-secure in realizing a functionality $\mathcal{F}$ if, for every probabilistic polynomial-time (PPT) adversary Adv targeting an execution of $\Pi$, there exists another PPT adversary known as a simulator Sim attacking the ideal execution of $\mathcal{F}$ such that the executions of $\Pi$ with Adv and that of $\mathcal{F}$ with Sim are indistinguishable to any PPT environment $\mathcal{Z}$.

**UC for private lookup evaluation.** Next, we prove the security of our private lookup table protocol $\Pi_{\mathsf{lookup}}$. We first provide the functionality for the private lookup table evaluation in Fig. 16.

*The idea world execution* $\mathsf{Ideal}_{\mathcal{F}_{\mathsf{lookup}},\mathsf{Sim},\mathcal{Z}}(1^\lambda)$. In the ideal world, the parties $\mathcal{P} := \{\mathcal{C}, \mathcal{S}\}$ only communicate with the ideal functionality $\mathcal{F}_{\mathsf{lookup}}$ with the executed function $f$. Both parties send their private data to $\mathcal{F}_{\mathsf{lookup}}$, and $\mathcal{F}_{\mathsf{lookup}}$ calculates and output the result to $\mathcal{C}$ and $\mathcal{S}$.

*The real world execution* $\mathsf{Real}_{\Pi_{\mathsf{lookup}},\mathsf{Adv},\mathcal{Z}}(1^\lambda)$. In the real world, the parties $\mathcal{P} := \{\mathcal{C}, \mathcal{S}\}$ communicate with each other, it executes the protocol $\Pi_{\mathsf{lookup}}$. Our protocols work in the

**Functionality $\mathcal{F}_{\mathsf{lookup}}$**

$\mathcal{F}_{\mathsf{lookup}}$ interacts with $\mathcal{C}$, $\mathcal{S}$ and the simulator Sim. Let $f$ denote the function encoded to the lookup table.

**Input:**

- Upon receiving $(\mathsf{Input}, \mathsf{sid}, (\mathcal{T}^{\ell_x,\ell_y}, x_1))$ from $\mathcal{S}$, record $(\mathcal{T}^{\ell_x,\ell_y}, x_1)$ and send $(\mathsf{Input}, \mathsf{sid}, \mathcal{S})$ to Sim, where $\mathcal{T}^{\ell_x,\ell_y} \in (\mathbb{Z}_{2^{\ell_y}})^{\ell_x}, x_1 \in \mathbb{Z}_{2^{\ell_x}}$.
- Upon receiving $(\mathsf{Input}, \mathsf{sid}, x_0)$ from $\mathcal{C}$, record $x_0$ and send $(\mathsf{Input}, \mathsf{sid}, \mathcal{C})$ to Sim, where $x_0 \in \mathbb{Z}_{2^{\ell_x}}$.

**Execution:**

- If both $\mathcal{T}^{\ell_x,\ell_y}, x_0, x_1$ are recorded, compute $y = \mathcal{T}^{\ell_x,\ell_y}(x_0 + x_1)$.
- Upon receiving $(\mathsf{Output}, \mathsf{sid}, y' \in \mathbb{Z}_{2^{\ell_y}})$ from Sim, if $\mathcal{S}$ is corrupted calculate $y_0 = y - y'$ and $y_1 = y'$, otherwise calculate $y_1 = y - y'$ and $y_0 = y'$.
- Send $(\mathsf{Output}, \mathsf{sid}, y_0)$ to $\mathcal{C}$ and $(\mathsf{Output}, \mathsf{sid}, y_1)$ to $\mathcal{S}$.

Fig. 16: The Ideal Functionality $\mathcal{F}_{\mathsf{lookup}}$ for private lookup table evaluation.

pre-processing model, but we analyze the offline and online protocols together as a whole.

**Theorem 1.** *Protocol $\Pi_{\mathsf{lookup}}$ UC-secure realizes functionality $\mathcal{F}_{\mathsf{lookup}}$ in the $\mathcal{F}_{(N-1,N)\text{-}\mathsf{OT}}$-hybrid model against semi-honest PPT adversaries with statical corruption, namely it holds:*

$$\mathsf{Real}_{\Pi_{\mathsf{lookup}},\mathsf{Adv},\mathcal{Z}}(1^\lambda) \approx \mathsf{Ideal}_{\mathcal{F}_{\mathsf{lookup}},\mathsf{Sim},\mathcal{Z}}(1^\lambda)$$

*Proof.* Before proving Theorem 1, we replace $\Pi_{(2^{\ell_x}-1,2^{\ell_x})\text{-}\mathsf{ROT}}$ with functionality $\mathcal{F}_{(2^{\ell_x}-1,2^{\ell_x})\text{-}\mathsf{ROT}}$ in Chase *et. al* [11]. To prove Theorem 1, we construct a PPT simulator $\mathcal{S}$, such that no non-uniform PPT environment $\mathcal{Z}$ can distinguish between the ideal world $\mathsf{Ideal}_{\mathcal{F}_{\mathsf{lookup}},\mathcal{S},\mathcal{Z}}(1^\lambda)$ and the real world $\mathsf{Real}_{\Pi_{\mathsf{lookup}},\mathsf{Adv},\mathcal{Z}}(1^\lambda)$. We consider the following cases:

Case 1: $\mathcal{C}$ is corrupted. We construct the simulator Sim which internally runs Adv, forwarding messages to/from $\mathcal{Z}$ and simulates the interface of honest $\mathcal{S}$.

- Upon receiving $(\mathsf{Input}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{lookup}}$, Sim starts simulation.
- Sim emulates $\mathcal{F}_{(N-1,N)\text{-}\mathsf{ROT}}$ and forward the output $\boldsymbol{m}_i \in (\mathbb{Z}_{2^{\ell_y}})^{2^{\ell_x}}$ for $i \in [2^{\ell_x}] \backslash \{[\![r]\!]_0\}$ and $[\![r]\!]_0$ to $\mathcal{C}$.
- Sim generates the matrix $\mathbf{M}$ by using the $\{\boldsymbol{m}_i\}_{i \in [2^{\ell_x}]}$ as the column vectors, and right circular shift the $i^{\text{th}}$ row of $\mathbf{M}$ by $i$ positions locally for $i \in [2^{\ell_x}]$.
- Sim computes $v_i = \sum_{j=0}^{2^{\ell_x}-1} m_{(i,j)}$ and $u_i = \sum_{j=0}^{2^{\ell_x}-1} -m_{(j,i)}$ to generate $\boldsymbol{u} := (u_0, \ldots, u_{2^{\ell_x}-1})$ and $\boldsymbol{v} := (v_0, \ldots, v_{2^{\ell_x}-1})$.
- Sim picks random vector $\boldsymbol{s}$ and acts as $\mathcal{S}$ to send $\boldsymbol{s}$ to $\mathcal{C}$.
- Sim picks random $[\![\delta]\!]_1^{\ell_x}$ and acts as $\mathcal{S}$ to send it to $\mathcal{C}$.
- Upon receiving $[\![\delta]\!]_0^{\ell_x}$ from $\mathcal{C}$, Sim extracts $[\![x]\!]_0^{\ell_x} = [\![\delta]\!]_0^{\ell_x} + [\![r]\!]_0^{\ell_x}$.
- Sim calculate $\delta := [\![\delta]\!]_0^{\ell_x} + [\![\delta]\!]_1^{\ell_x}$ and $[\![\mathcal{T}']\!]_0$ with the values $\boldsymbol{s}, [\![r]\!]_0^{\ell_x}, \boldsymbol{w}$.

TABLE V: Online performance and offline performance in C/S setting for CNN model comparison with CryptoFlow2 [56].(We take $\ell = 64$ and $\ell' = 8$. The bandwidth is 377 MBps and 40 MBps in the LAN and the WAN setting respectively and the latency is 0.3ms and 80ms respectively. )

| Model | Protocol | LAN | WAN | Comm. |
|---|---|---|---|---|
| ImageNet_ SqueezeNet | CryptoFlow2 | 44.3s | 293.6s | 26.07GB |
| | Our Online | 1.66s | 5.88s | 0.077GB |
| | Our Offline | 80.54s | 440.20s | 108.0GB |
| ImageNet_ ResNet50 | CryptoFlow2 | 619.4s | 3611.6s | 370.8GB |
| | Our Online | 7.32s | 13.46s | 0.45GB |
| | Our Offline | 352.40s | 1586.59s | 481.8GB |

- Sim inputs $(\mathsf{Input}, \mathsf{sid}, [\![x]\!]_0^{\ell_x})$ to $\mathcal{F}_{\mathsf{lookup}}$.
- Sim inputs $(\mathsf{Output}, \mathsf{sid}, [\![\mathcal{T}'(\delta)]\!]_0)$ to $\mathcal{F}_{\mathsf{lookup}}$.

Informally, we discuss the indistinguishable. Obviously, in the above simulation, considering $\mathcal{S}$ will input $[\![x]\!]_1^{\ell_x}$ to $\mathcal{F}_{\mathsf{lookup}}$ calculates $y = \mathcal{T}([\![x]\!]_0^{\ell_x} + [\![x]\!]_1^{\ell_x})$ and output $y - [\![\mathcal{T}'(\delta)]\!]_0$ to $\mathcal{S}$. When $\mathcal{C}$ sets output as $[\![\mathcal{T}'(\delta)]\!]_0$, we get the same output in the real world. To illustrate the indistinguishable of temporary value, we prove that the ideal world $[\![\delta]\!]_1^{\ell_x}$, $\boldsymbol{s}$ are generated randomly. For $\boldsymbol{s}$, it is easy to see that it is uniform random in the real world since the vector $\boldsymbol{u}$ can be viewed as a random vector. So the values $[\![\delta]\!]_1^{\ell_x}$, $\boldsymbol{s}$ both keep the same distribution between the real world and ideal world and can not be distinguished.

Case 2: $\mathcal{S}$ is corrupted. We construct the simulator Sim which internally runs Adv, forwarding messages to/from $\mathcal{Z}$ and simulates the interface of honest $\mathcal{C}$.

- Upon receiving $(\mathsf{Input}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{lookup}}$, Sim starts simulation.
- Sim emulates $\mathcal{F}_{(N-1,N)\text{-}\mathsf{ROT}}$ and forward the output $\boldsymbol{m}_i \in (\mathbb{Z}_{2^{\ell_y}})^{2^{\ell_x}}$ for $i \in [2^{\ell_x}]$ to $\mathcal{C}$.
- Sim calculate $\boldsymbol{v}, \boldsymbol{u}, \boldsymbol{w}$ using the output of $\mathcal{F}_{(N-1,N)\text{-}\mathsf{ROT}}$.
- Upon receiving $\boldsymbol{s}$ for $\mathcal{S}$, Sim extracts $\hat{\mathcal{T}} = \boldsymbol{s} - \boldsymbol{u}$.
- Sim picks random $[\![\delta]\!]_0^{\ell_x}$ and acts as $\mathcal{C}$ to send it to $\mathcal{S}$.
- Upon receiving $[\![\delta]\!]_1^{\ell_x}$ from $\mathcal{S}$, Sim calculates $\delta = [\![\delta]\!]_0^{\ell_x} + [\![\delta]\!]_1^{\ell_x}$.
- Sim inputs $(\mathsf{Input}, \mathsf{sid}, (\hat{\mathcal{T}}, [\![\delta]\!]_1^{\ell_x}))$ as like $\mathcal{S}$ to $\mathcal{F}_{\mathsf{lookup}}$.
- Sim inputs $(\mathsf{Output}, \mathsf{sid}, v_\delta)$ to $\mathcal{F}_{\mathsf{lookup}}$.

Informally, we discuss the indistinguishable. For the output, in above simulation, $\mathcal{F}_{\mathsf{lookup}}$ will calculate $y = \hat{\mathcal{T}}([\![\delta]\!]_1^{\ell_x} + [\![x]\!]_0^{\ell_x})$. Since $\hat{\mathcal{T}}(x) = \mathcal{T}(x + [\![r]\!]_1^{\ell_x})$ and $[\![\delta]\!]_1^{\ell_x} = [\![x]\!]_1^{\ell_x} - [\![r]\!]_1^{\ell_x}$, it equals to $\mathcal{T}([\![x]\!]_1^{\ell_x} - [\![r]\!]_1^{\ell_x} + [\![x]\!]_0^{\ell_x} + [\![r]\!]_1^{\ell_x})$, which is $y = \mathcal{T}(x)$. $\mathcal{F}_{\mathsf{lookup}}$ sends $y - v_\delta$ to $\mathcal{C}$, while the corrupted $\mathcal{S}$ hold $v_\delta$ due to $\delta$ received from Sim. Furthermore, $[\![\delta]\!]_0^{\ell_x}$ in the ideal world is randomly generated which is indistinguishable from the real world.

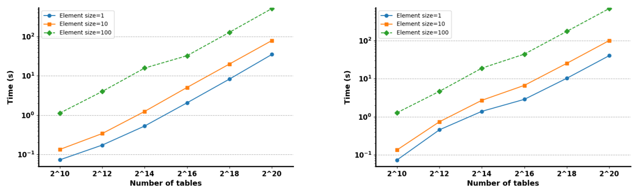
This concludes the proof. □

## B. Other benchmarks

In this section, we give more benchmarks. We set bandwidth 5Gbps/40Mbps for the LAN and the WAN settings respectively and the latency 0.05 ms / 60 ms respectively.

**Performance for C/S setting Compared to CrypTFlow2 [56].** Table V depicts the performance comparison between our framework and CrypTFlow2. Our framework online phase is 300× faster than that ofCrypTFlow2 on ResNet50.

**Offline performance for C/S setting.** Figure 17 shows the offline performance for the C/S setting with different element sizes in LAN and WAN settings. The element size in the legend represents the number of elements in each line of the lookup table. While the element size is 1, the lookup table can be used to compute functions such as ReLU or Maxpool for individual elements. However, when the element size is 10, 100, or higher, these lookup tables are employed for matrix multiplication with dimensions of 10, 100, or higher. During matrix multiplication, the same input is multiplied with multiple elements in one row or column, allowing these tables to be efficiently combined into a single table line for offline processing. As the number of tables and the element size increase, the corresponding runtime also increases. Despite some fluctuations, the growth in time is largely linearly related to the increase in the number of tables. Additionally, the time used for WAN does not exhibit a significant increase compared to LAN time. This means that during this runtime process, network communication does not significantly contribute to the overall time consumption. Given our device and code limitations, there is theoretically room for further improvement in runtime, especially considering that many computations can be parallelized and better memory management.



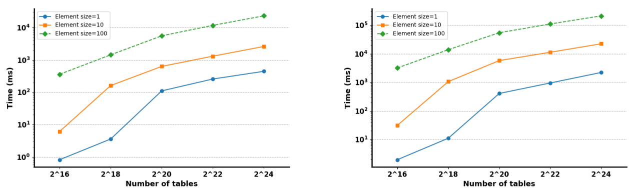
(a) LAN (b) WAN

Fig. 17: The running time of offline phase for C/S setting compared with different element sizes in LAN and WAN setting.

**Offline performance for outsourcing setting.** Figure 18 shows the offline performance for outsourcing settings with different element sizes in LAN and WAN settings. Due to the impact of factors such as network and memory, the initial part of the curve exhibits slight fluctuations. However, it still demonstrates a linear relationship. In the context of outsourcing, two computational nodes primarily handle the lookup table from the server, while the server providing the machine learning model performs relatively less complex computations compared with the C/S setting. Notably, the impact of network communication restrictions is more pronounced in this scenario, as the WAN time significantly exceeds the LAN time.

## C. *Operator for CNN and Transformer*

**Convolutional Neural Network.** We implemented typical convolutional neural network (CNN) models in our framework,



(a) LAN (b) WAN

Fig. 18: The running time of offline phase for outsourcing setting compared with different element sizes in LAN and WAN setting.

including LeNet, VGG16, and ResNet18. All SISO-type operators, such as batch normalization, ReLU, sigmoid, and so on, can be directly evaluated using our paradigm. For the MISO-type operators, we discuss their implementation as follows.

- Convolution(Conv)/Fully Connection(FC)/General Matrix Multiplication(GeMM): The convolution operator can be transferred to the general matrix multiplication. For instance, the naive method explicitly lowers the convolution to GeMM, commonly known as im2col. Furthermore, the full connection can also be represented as the GeMM form. As mentioned before, viewing the weight as part of the lookup table, our framework can deal with GeMM as a "multiple lookup table query with a single index". For the sum part of GeMM, due to each item keeping the same scale factor, it can be calculated by the sum of each quantized value. In particular, we take two-dimensional multiplication as an example. Considering $z = z_1 + z_2 = x_1 \cdot y_1 + x_2 \cdot y_2$, where $x_1, x_2$ are in the same vector so that they keep the same scale factor $s_1, b_1$, the same to $y_1, y_2$ with scale factor $s_2, b_2$. It holds that $z' = \frac{s_1 \cdot s_2}{s_3}((x'_1 - b_1)(y'_1 - b_2) + (x'_2 - b_1)(y'_2 - b_2)) + b_3$. Considering $z'_1 = \frac{s_1 \cdot s_2}{s_3}(x'_1 - b_1)(y'_1 - b_2) + b_3$ and $z'_2 = \frac{s_1 \cdot s_2}{s_3}(x'_2 - b_1)(y'_2 - b_2) + b_3$, we have $z' = z'_1 + z'_2 - b_3$. Without loss of generality, for $n$-dimension inner product, the quantized output holds $z' = \sum_{i=0}^{N-1} z'_i - (n-1) \cdot b_3$. For the secret form $[z']^{\ell'} = \sum_{i=0}^{N-1} [z'_i]^{\ell'} - (n-1) \cdot b_3$, the part minus $(n-1) \cdot b_3$ can be evaluated locally by $\mathcal{S}$. Note that since matrix multiplication requires summing during the output process and considering the precision requirements after summing, we use higher precision outputs, such as 16-bit, to generate the lookup tables when performing matrix multiplication.
- Max Pooling or Average Pooling: The max pooling is an expensive operator in our framework since evaluating $n$-dimensional max pooling is equivalent to performing $n - 1$ comparisons. We must convert the dual-input operator to a SISO structure to evaluate the comparison. For example, for 8-dimension max pooling with 8-bit output, we let the previous layer output 4-bit vector, such as $[x_0]^4, \ldots, [x_7]^4$. We evaluate the comparison between each share and obtain 4-dimension 4-bit shared vector. We perform comparison layer by layer until the dimension is reduced to 1. In the last layer, we utilize the 8-bit output lookup table to recover 8-bit quantization.

The average pooling is much cheaper, the sum part of it can be evaluated by locally adding the quantized value since all the elements are in the same vector, which keeps the same scale factor. To avoid a wrap-around of the secret, we need to perform division before summation( with higher precision output). For the division, we use the SISO lookup table.

**Large Language Models.** For the Large Language Models (LLM), we implement GPT-2 models. In LLM, the GeLU operator is SISO-type, allowing us to apply our paradigm directly. For the matrix multiplication which all the input is unknown to the model server, we convert the dual-input operator to the SISO structure. For the softmax and layer normalization, we discuss the implementation as follows.

- Softmax: For the softmax operator $\mathsf{Softmax}(\boldsymbol{x}) := (\frac{e^{x_i}}{\sum_{i=0}^{n-1} e^{x_i}})_{i\in[n]}$ (The scale factor of $\boldsymbol{x}$ is $\hat{s}$ and $\hat{b}$), we follow the typical PPML implementation [21] and modify it for quantization scheme. We first perform $\mathsf{max}(\boldsymbol{x})$ to find out the max element of $\boldsymbol{x}$, denoted by $\hat{x}$. After that we define temporary variable $y_i = e^{x_i-\hat{x}} = e^{s(x'_i-b-\hat{x}+b)}$. Considering $e^{x'_i-\hat{x}} \le 1$, we can take the scale factor $\hat{s}$ and $\hat{b}$ for $\boldsymbol{y} := (y_0,\dots,y_{n-1})$ as $\hat{s}\cdot 2^{\ell'}/n = 1$ and $\hat{b} = 0$, such that $\sum_{i=0}^{n-1} y'_i$ will not wrap around. We have $\hat{s}\cdot y'_i = e^{s(x'_i-\hat{x})}$ and $\frac{e^{x_i}}{\sum_{i=0}^{n-1} e^{x_i}} = \frac{y'_i}{\sum_{i=0}^{n-1} y'_i}$. Based on these logics, we evaluate the $\mathsf{Softmax}(\boldsymbol{x})$ as follows. i. evaluate $y'_i = \frac{1}{\hat{s}}\cdot e^{s(x'_i-\hat{x})}$ with lookup table. ii. calculate $\hat{y} = \sum_{i=0}^{n-1} y'_i$ in quantized value. iii. perform $z'_i = \frac{1}{s_2}\cdot\frac{y'_i}{\hat{y}} + b_2$ with lookup table to obtain the quantized output, where $s_2, b_2$ is the scale factor for $\boldsymbol{z} := (z_0,\dots,z_{n-1})$.
- Word2Vec/Gather: Word2Vec is used for LLM tasks to produce word embeddings, which are dense vector representations of words. The Word2Vec is realized by the operator Gather, which picks a row vector from a matrix. For GPT-2, the word matrix has 50257 row vectors for each potential word. Each row vector involves 768 quantized elements in GPT-2-base or 1600 quantized elements in GPT-2-XL. We utilize our lookup table to evaluate Gather, where the lookup table input is in $\mathbb{Z}_{50257}$ and the output is in $\mathbb{Z}_{2^{\ell'}}$.

## D. Related Works

The current works on secure multi-party computation in PPML mainly focus on two-party, three-party, and four-party settings. The representative works for two-party setting are SecureML [50], Delphi [48], Chameleon [58], CryptFlow2 [56], ABY2.0 [53], Cheetah [30], and Li et al. [45]. For three-party setting, there are SecureNN [64], Falcon [65], ABY$^3$ [49], ASTRA [12], BLAZE [54], CryptFlow [41]. SWIFT [51], FLASH [9], and Trident [13] are considering four-party setting. Recently, Orca [33] applied function secret sharing on the PPML where the three parties share the function secret share key in the offline phase. and the two parties perform secure computation in the online phase. Edabits [23] implements PPML in multiparty settings and supports the security against malicious adversaries. For the GPU setting, CryptGPU [62],CrypTen [40], Piranha [40] and Orca [33] construct the GPU platform for PPML. Recently, THE-X [14], MPCformer [44], Privformer [5], BOLT [52], BumbleBee [36] and Sigma [26], Grotto [61] focus on the large language models and also use GPU to accelerate performance. Note that Sigma utilizes the lookup table evaluation to perform non-linear function evaluation.

Using quantization techniques in the field of PPML [72], [42], [47], [70], [43], [60] is a promising approach to improving PPML performance, with the most typical application being in distributed machine learning, primarily constructed based on federated learning. Currently, the use of quantization techniques for evaluating PPML in MPC scenarios is still in its early stages. A series of studies employ traditional MPC methods to evaluate quantized models (Riazi *et. al* [57], Agrawal *et. al* [4] and Keller *et. al* [38]). To ensure performance, they use public and fixed quantization parameters set as powers of 2, which sacrifices model accuracy, and the public quantization parameters can potentially leak model information."

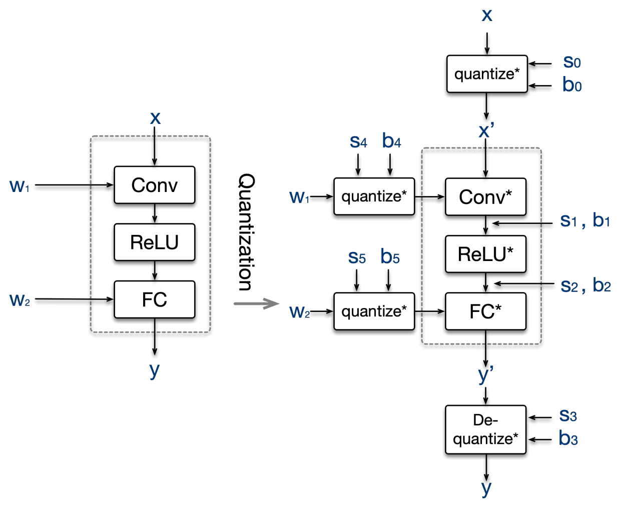


Fig. 19: Case of quantization.

## E. Quantization Model Diagram

Fig. 19 illustrates a quantization workflow for a neural network model. On the left, we have an unquantized model, which takes an input $x$ and processes it through convolution, ReLU and Fully Connection layers. On the right, the quantized model is shown in detail:

1) The input $x$ is quantized into $x'$ using a quantization function with scale $s_0$ and offset $b_0$.
2) Unquantized weights $w_1$ and $w_2$ are processed with additional quantization parameters $s_4$, $b_4$, $s_5$, and $b_5$.
3) The quantized input $x'$ is passed through quantized versions of the layers: Conv*, ReLU*, and FC*, and generate the output $y'$.
4) The output of the quantized model, $y'$, is de-quantized using parameters $s_3$ and $b_3$ to obtain the final output $y$.