Faster Lookup Table Evaluation with Application to Secure LLM Inference

Xiaoyang Hou, Zhejiang University, xiaoyanghou@zju.edu.cn
Jian Liu[✉], Zhejiang University, liujian2411@zju.edu.cn
Jingyu Li, Zhejiang University, jingyuli@zju.edu.cn
Jiawen Zhang, Zhejiang University, kevinzh@zju.edu.cn
Kui Ren, Zhejiang University, kuiren@zju.edu.cn

***Abstract*—As large language models (LLMs) continue to gain popularity, concerns about user privacy are amplified, given that the data submitted by users for inference may contain sensitive information. Therefore, running LLMs through secure two-party computation (a.k.a. secure LLM inference) has emerged as a prominent topic. However, many operations in LLMs, such as Softmax and GELU, cannot be computed using conventional gates in secure computation; instead, lookup tables (LUTs) have to be utilized, which makes LUT to be an essential primitive in secure LLM inference.**

In this paper, we propose ROTL, a secure two-party protocol for LUT evaluations. Compared with FLUTE (the state-of-the-art LUT presented at Oakland '23), it achieves upto 11.6× speedup in terms of overall performance and 155× speedup in terms of online performance. Furthermore, ROTL can support arithmetic shares (which is required by secure LLM inference), whereas FLUTE can only support boolean shares. At the heart of ROTL is a novel protocol for secret-shared rotation, which allows two parties to generate additive shares of the rotated table without revealing the rotation offset. We believe this protocol is of independent interest. Based on ROTL, we design a novel secure comparison protocol; compared with the state-of-the-art, it achieves a 2.4× bandwidth reduction in terms of online performance.

To support boolean shares, we further provide an optimization for FLUTE, by reducing its computational complexity from $O(l \cdot n^2)$ to $O(n \log n + l \cdot n)$ and shifting $O(n \log n)$ computation to the preprocessing phase. As a result, compared with FLUTE, it achieves upto 10.8× speedup in terms of overall performance and 962× speedup in terms of online performance.

I. Introduction

With the increasing popularity of large language models (LLMs), there is an amplified concern about user privacy, given that the data provided by users for inference may contain sensitive information. Hence, *Secure inference* [10], [21], [20], [23], [30], [28], [17], [14] has emerged as a prominent topic, which runs the inference stage in a way such that the server ($\mathcal{S}$) learns nothing about clients' input and a client ($\mathcal{C}$) learns nothing about the model except the inference results. Roughly, it can be considered as a secure two-party computation (2PC) protocol that is customized for model inference.

However, many operations in LLMs, such as softmax and GELU, cannot be computed using conventional gates in 2PC; instead, lookup tables (LUTs) have to be utilized, which makes LUT to be an essential primitive in secure LLM inference. A LUT protocol allows two parties, holding a secret-shared index i, to learn x_i from a public table $\mathbf{x} \in \mathbb{Z}_{2^l}^n$. A *preprocessing phase* is usually introduced to prepare some expensive and input-independent work so that the *online phase* can be done efficiently.

The most common way for LUT evaluation is based on 1-out-of-n OT. Specifically, $\mathcal{S}$ generates a LUT output for each of $\mathcal{C}$'s n possible input shares, and masks these outputs with a single random number, which is $\mathcal{S}$'s output share. Then, $\mathcal{C}$ uses 1-out-of-n OT to get its own output share. Although this protocol is computationally efficient, it has to transfer the whole table during online phase.

The state-of-the-art LUT protocol (named FLUTE) avoids transferring the whole table by converting the LUT description into boolean expressions, the circuit of which is then evaluated as a multi-fan-in inner product [4]. However, FLUTE involves expensive computations in the online phase, due to the evaluation of the multi-fan-in AND gates. Moreover, FLUTE can only support boolean shares for its input and output, whereas secure LLM inference desires arithmetic shares as it involves massive matrix multiplications. To be used in secure LLM inference, FLUTE has to be augmented with both boolean-to-arithmetic (B2A) and arithmetic-to-boolean (A2B) conversions, and the A2B conversion is particularly expensive.

Our contribution. In this paper, we propose ROTL, a LUT protocol that is significantly faster than FLUTE and can support arithmetic shares. The rough idea of ROTL is to have $\mathcal{C}$ and $\mathcal{S}$ jointly right-rotate the table $\mathbf{x}$ for s elements in the preprocessing phase, with both the rotated table and s being secret-shared between $\mathcal{C}$ and $\mathcal{S}$. Then, in the online phase, $\mathcal{C}$ and $\mathcal{S}$ can simply recover $(i+s)$ and output the $(i+s)$-th element in the rotated table. This idea aligns with the approaches presented by OTTT [19] and OP-LUT [9]. However, both OTTT and OP-LUT require expensive circuit evaluations to rotate the table: OTTT evaluates a boolean circuit representing the table for every possible input, and OP-LUT can be considered as a natural generalization of the GMW protocol. In contrast, we propose a novel protocol for table rotation that is significantly more lightweight. Recognizing that rotation is a special kind of permutation, we leverage the secret-shared permutation protocol proposed by Chase et al. [5], which is already quite lightweight, demanding only $n \log n$ random-OTs. By harnessing the inherent characteristics of rotation, we achieve a significant reduction in the number

[✉]Jian Liu is the corresponding author.

of necessary random-OTs, cutting it down to a mere $\log n$. Furthermore, we come up with a way allowing $\mathcal{C}$ and $\mathcal{S}$ to rotate a selection vector $\mathbf{b} \in \mathbb{Z}_2^n$ rather than rotating the table $\mathbf{x} \in \mathbb{Z}_{2^l}^n$. As a result, we make the communication overhead independent of l.

While boolean shares are less commonly employed for LUTs, they find application in the evaluation of boolean circuits. For completeness, we introduce another LUT solution for boolean shares named FLUTE+. It can be considered as an optimization of FLUTE, by reducing the computational complexity of FLUTE from $O(l \cdot n^2)$ to $O(n \log n + l \cdot n)$ and shifting $O(n \log n)$ computation to the preprocessing phase.

In addition to LUTs, *secure comparison* is another critical primitive for secure LLM inference, extensively employed in operations such as truncation, softmax, and GELU. Based on ROTL, we introduce a novel secure comparison protocol, strategically shifting the main overhead to the preprocessing phase. Compared with the state-of-the-art [**?**], it achieves a xxx speedup in terms of online performance.

We summarize our contributions as follows:

- A novel protocol for secret-shared rotation (Section III);
- A novel LUT protocol (named ROTL), which is upto $155\times$ faster than FLUTE and supports arithmetic shares (Section IV);
- An optimization of FLUTE (named FLUTE+), achieving upto $962\times$ speedup (Section V);
- A novel secure comparison protocol, which achieves a $2.4\times$ bandwidth reduction over the state-of-the-art [30].
- An application of ROTL to secure LLM inference (Section VII-E);
- A full-fledged implementation and comprehensive benchmark (Section VII).

TABLE I: A table of frequent notations.

Notation	Description
$\mathcal{C}$	client
$\mathcal{S}$	server
λ	security parameter
n	table length
l	bit-length of each element in the table
s	rotation offset
$\langle x \rangle^l$	$(\langle x \rangle^l_{\mathcal{S}}, \langle x \rangle^l_{\mathcal{C}})$ s.t. $x = \langle x \rangle^l_{\mathcal{S}} + \langle x \rangle^l_{\mathcal{C}} \mod 2^l$
$\mathcal{F}_{\mathsf{LUT}}$	ideal functionality for lookup table evaluation
$\mathcal{F}_{\mathsf{Rotate}}$	ideal functionality for secret-shared rotation
$\mathcal{F}_{\mathsf{Mult}}$	ideal functionality for secret-shared multiplication
$\mathcal{F}_{\mathsf{AND}}$	ideal functionality for secret-shared AND
$\mathcal{F}_{\mathsf{CMP}}$	ideal functionality for comparison $b \leftarrow \mathsf{CMP}(x, y)$: $b = 1$ if $x \geq y$; $b = 0$ otherwise
$\mathcal{F}_{\mathsf{MUX}}$	ideal functionality for multiplexer $y \leftarrow \mathsf{MUX}(x, b)$: $y = x$ if $b = 1$; $y = 0$ if $b = 0$

II. Background and Preliminaries

In this section, we present the necessary background and preliminaries for this paper.

A. Notations

We use $\langle x \rangle^l = (\langle x \rangle^l_{\mathcal{S}}, \langle x \rangle^l_{\mathcal{C}})$ to denote 2-out-of-2 additive secret-sharing over $\mathbb{Z}_{2^l}$. Namely, $x = \langle x \rangle^l_{\mathcal{S}} + \langle x \rangle^l_{\mathcal{C}} \mod 2^l$. For simplicity, we omit the l notation of $\langle x \rangle^l$ when it is not contextually relevant. We denote vectors with bold fonts and elements inside a vector with indices. For example, $\mathbf{v}$ is a vector of n elements and v_i is the i-th element in $\mathbf{v}$. As our target scenario – secure LLM inference – proceeds in a client-server setting, we use $\mathcal{C}$ and $\mathcal{S}$ to denote the two parties in all protocols. We use Π to denote a protocol and use $\mathcal{F}$ to denote the ideal functionality of a protocol. We use $\mathsf{view}^{\mathcal{C}}_{\Pi}$/$\mathsf{view}^{\mathcal{S}}_{\Pi}$ to denote the view of $\mathcal{C}$/$\mathcal{S}$ when they run the protocol Π.

Table I provides a summary of the frequently used notations in this paper.

B. Lookup table

The ideal functionality of a lookup table (LUT) protocol $\mathcal{F}_{\mathsf{LUT}}$ takes a public table $\mathbf{x} \in \mathbb{Z}_{2^l}^n$ and a secret-shared index $i \in \mathbb{Z}_n$ as inputs, and returns the secret-shared x_i. Figure 1 describes this functionality. In this paper, our primary focus is on LUT with arithmetic shares, as this configuration is pertinent to secure LLM inference. The discussion on LUTs with boolean shares will be presented in Section V.

Functionality $\mathcal{F}_{\mathsf{LUT}}$

Parameter: $\mathbf{x} \in \mathbb{Z}_{2^l}^n$
Input:

- $\mathcal{C}$: $\langle i \rangle_{\mathcal{C}} \in \mathbb{Z}_n$
- $\mathcal{S}$: $\langle i \rangle_{\mathcal{S}} \in \mathbb{Z}_n$

Output:

- $\mathcal{C}$: $\langle x_i \rangle_{\mathcal{C}} \in \mathbb{Z}_{2^l}$
- $\mathcal{S}$: $\langle x_i \rangle_{\mathcal{S}} \in \mathbb{Z}_{2^l}$

Fig. 1: Ideal functionality for LUT.

C. Puncturable pseudorandom function (PPRF)

A *puncturable pseudorandom function* (PPRF) allows one with a master key to evaluate a PRF at all points of its domain; allows one with a *punctured* key to evaluate the PRF at all points except a punctured point. A PPRF can be used to efficiently achieve $(n-1)$-out-of-n random OT, which, on input $i \in \mathbb{Z}_n$ from $\mathcal{S}$, allows $\mathcal{S}$ and $\mathcal{C}$ to jointly generate a vector $\mathbf{v}$ with random-looking elements, s.t., $\mathcal{S}$ obtains all elements in $\mathbf{v}$ except for v_i (denoted by $\mathbf{v}^{\mathcal{S}}$), and $\mathcal{C}$ obtains the whole vector $\mathbf{v}$ (denoted by $\mathbf{v}^{\mathcal{C}}$) without learning the index i. Specifically, it has the following properties:

- *Correctness:* $v^{\mathcal{C}}_j = v^{\mathcal{S}}_j \ \forall \ j \neq i$.
- *Position hiding:* a compromised $\mathcal{C}$, who, in addition to its view in the protocol execution, receives two distinct

indices $i, i' \in \mathbb{Z}_n$, cannot differentiate between the following two executions:
  - where $\mathcal{S}$ uses i as its input;
  - where $\mathcal{S}$ uses i' as its input.
- *Value hiding.* a compromised $\mathcal{S}$, who, in addition to its view in the protocol execution, receives the vector $\mathbf{v}^{\mathcal{C}}$, cannot differentiate between the following two executions:
  - where $\mathbf{v}^{\mathcal{C}}$ is generated according to PPRF;
  - where $\mathbf{v}^{\mathcal{C}}$ is generated according to PPRF, but $v_i^{\mathcal{C}}$ is replaced with a random element from the domain.

The PPRF-based $(n-1)$-out-of-n random OT only requires $\log n$ parallel executions of 1-out-of-2 OTs. In this paper, we abuse the notion and use PPRF to denote the PPRF-based $(n-1)$-out-of-n random OT.

D. Secret-shared permutation

Chase et al. [5] propose a protocol for secret-shared permutation, which allows two parties to learn secret-shares of a permutated array $\mathbf{x} \in \mathbb{Z}_{2^l}^n$. Figure 2 shows the ideal functionality for this protocol: it takes an array from $\mathcal{C}$ and a permutation π from $\mathcal{S}$, and returns secret-shares of a permutated array. By $\pi(x)$, we denote the permuted vector $(x[\pi(1)], ..., x[\pi(N)])$. Their protocol also works for the case when $\mathbf{x}$ was secret-shared between two parties (instead of being an input of one party).

Functionality $\mathcal{F}_{\mathsf{Permut}}$

Input:
- $\mathcal{C}$: $\mathbf{x} \in \mathbb{Z}_{2^l}^n$
- $\mathcal{S}$: a permutation π

Output:
- $\mathcal{C}$: $\mathbf{r}$
- $\mathcal{S}$: $\mathbf{r} \oplus \pi(\mathbf{x})$, where $\mathbf{r} \in_{\$} \mathbb{Z}_{2^l}^n$

Fig. 2: Ideal functionality for secret-shared permutation.

The protocol is described in Figure 3. After Step 1, $\mathcal{C}$ learns a matrix $[\mathbf{u}_0, ..., \mathbf{u}_{n-1}]$, and $\mathcal{S}$ learns the same matrix except for elements corresponding to the permutation, i.e., $\mathcal{S}$ learns nothing about $[u_{0,\pi(0)}, ..., u_{n-1,\pi(n-1)}]$. Figure 4 visualizes these two matrices. In Step 2, $\mathcal{C}$ sets $\mathbf{r}$, $\mathbf{a}$ to be row- and column-wise sums of the matrix elements. In Step 3, $\mathcal{S}$ computes each c_i by taking the sum of row i and adding the sum of column $\pi(i)$. Notice that $c_i = a_{\pi(i)} \oplus r_i$. Then, $\mathcal{S}$'s output in Step 5 is:

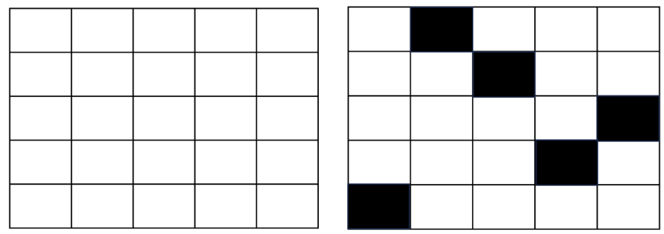
$$\pi(\mathbf{x} \oplus \mathbf{a}) \oplus \mathbf{c} = \pi(\mathbf{x}) \oplus \pi(\mathbf{a}) \oplus \pi(\mathbf{a}) \oplus \mathbf{r} = \pi(\mathbf{x}) \oplus \mathbf{r}.$$

That means $\mathcal{S}$ and $\mathcal{C}$ secret-share $\pi(\mathbf{x})$ at the end of the protocol. In [5], they use a Benes permutation network to optimize the computational complexity of Step-2 and 3 when n is large. We omit this optimization as $n \leq 256$ in our target scenario (cf. Section II-E).

Protocol Π_{Permut}

1) $\mathcal{C}$ and $\mathcal{S}$ run n executions of PPRF in parallel, where $\mathcal{S}$ uses $\pi(i)$ as its input in execution i, for $i \in \mathbb{Z}_n$. Let $\mathbf{u}_i$ and $\mathbf{v}_i$ be the outputs of the i-th execution, for $\mathcal{C}$ and $\mathcal{S}$ respectively ($\mathcal{S}$ fills the punctured positions in $\mathbf{v}_i$ with 0s).
2) For $i \in \mathbb{Z}_n$, $\mathcal{C}$ sets $r_i := \bigoplus_j u_{i,j}$, $a_i := \bigoplus_j u_{j,i}$.
3) For $i \in \mathbb{Z}_n$, $\mathcal{S}$ sets $c_i := (\bigoplus_j v_{i,j}) \oplus (\bigoplus_j v_{j,\pi(i)})$.
4) $\mathcal{C}$ sends $\mathbf{x} \oplus \mathbf{a}$ to $\mathcal{S}$ and outputs $\mathbf{r}$.
5) $\mathcal{S}$ outputs $\pi(\mathbf{x} \oplus \mathbf{a}) \oplus \mathbf{c}$.

Fig. 3: The secret-shared permutation protocol.



(a) $\mathcal{C}$ receives the full matrix.

(b) $\mathcal{S}$ receives a "punctured" matrix, where the missing elements are at positions $(i, \pi(i))$.

Fig. 4: Visualization of the two matrices received by $\mathcal{C}$ and $\mathcal{S}$ after Step 1 of Π_{Permut} (taken from [5]).

We remark that the data in Step 4 can be transferred together with PPRF in Step 1, to save one round of communication. The total communication cost of this protocol is $(2\lambda n \log n + nl)$ bits and only requires symmetric-key operations.

E. Secure LLM Inference

Suppose $\mathcal{S}$ holds a model and $\mathcal{C}$ holds an input $\mathbf{x}$. *Secure inference* [10], [21], [20], [23], [30], [28], [17], [14] runs the inference stage in a way such that $\mathcal{S}$ learns nothing about $\mathcal{C}$'s input and a $\mathcal{C}$ learns nothing about the model except the inference results. The input $\mathbf{x}$ to the model is typically undergone a left-shift by L bits (from floating-point to fixed-point), leading to l bits in total.

$\mathcal{F}_{\mathsf{LUT}}$ has been used extensively in secure LLM inference. As reported by [16], softmax and GELU occupy 43% computational cost and 54.8% communication cost of a secure GPT-2 inference, and $\mathcal{F}_{\mathsf{LUT}}$ is the main component of these operations.

Softmax. Softmax takes a secret-shared vector $\langle \mathbf{x} \rangle$ (with $\mathbf{x} \in \mathbb{Z}_{2^l}^m$) as input and normalizes each element x_i as follows:

$$\mathsf{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}.$$

The outputs add up to 1 and form a probability distribution. Hou et al. [16] provide a way to securely compute softmax as follows:

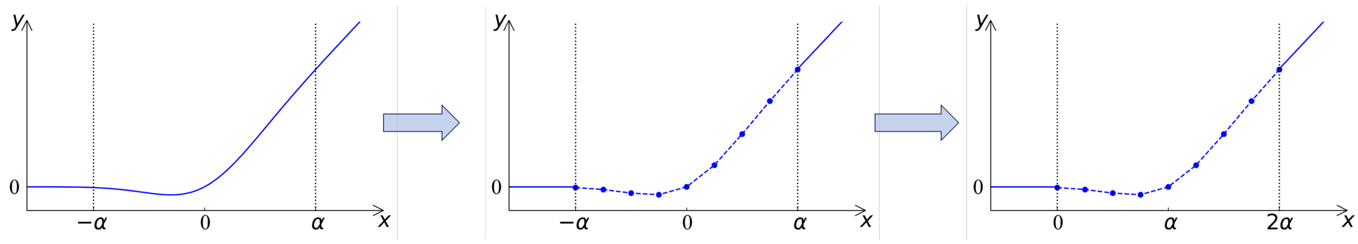


Fig. 5: GELU transformation (taken from [16]).

1) Each x_j is transformed to: $x'_j := x_j - \max(\mathbf{x})$.
2) Compute the exponential $e^{x'_j}$ for each x'_j:
   a) assume $x'_j \in \left[-16 \cdot 2^L, 0\right]$ and use $\mathcal{F}_{\mathsf{LUT}}$ to compute $e^{x'_j}$;
   b) use $\mathcal{F}_{\mathsf{CMP}}$ to compare x'_j with -16×2^l and use $\mathcal{F}_{\mathsf{MUX}}$ to set $e^{x'_j}$ to 0 if $x'_j < -16 \times 2^l$.
3) Compute $sum := \sum_{j=1}^{n} e^{x_j}$.
4) Use $\mathcal{F}_{\mathsf{LUT}}$ to compute the reciprocal: $\frac{1}{sum}$.
5) Use $\mathcal{F}_{\mathsf{Mult}}$ to compute $\frac{e^{x'_j}}{sum}$.

The LUT for computing exponential and reciprocal are with 2^{16} and 2^8 entries respectively. In particular, to compute the exponential, they use two LUTs where the first processes the upper 8 bits and the second processes the lower 8 bits; the final result is computed by multiplying the two looked up values from the two LUTs.

GELU. The most commonly used activation function in a LLM is GELU:

$$\mathsf{GELU}(x) = 0.5x(1 + \mathsf{Tanh}\left[\sqrt{2/\pi}(x + 0.044715x^3)\right]),$$

where $\mathsf{Tanh}(x) = 2\mathsf{Sigmoid}(2x) - 1$ and $\mathsf{Sigmoid}(x) = \frac{1}{1+e^{-x}}$. Figure 5 (left) depicts the original curve of $y = \mathsf{GELU}(x)$. It begins at zero for small values of x, and starts deviating from zero when x is around $-\alpha$. As x increases further, $\mathsf{GELU}(x)$ progressively approximates the linear function $y = x$.

Hou et al. [16] divide the curve into three large intervals:

- $y = 0$ when $x < -\alpha$;
- $y = \mathsf{GELU}(x)$ when $-\alpha \leq x \leq \alpha$;
- $y = x$ when $x > \alpha$.

The computation of the first and third intervals is straightforward. For the second interval, they divide the second interval into several small intervals and use a linear function ($y = ax + d$) to approximate the curve within each small interval, s depicted in Figure 5 (middle). Then, they right-shift the entire curve by α as shown in Figure 5 (right), after which the second interval becomes $[0, 2\alpha]$ and can be computed with a single LUT of 2^8 entries.

Since all such functions within a LLM involve lookup tables of sizes no larger than 256, our primary focus is on designing LUT protocols with $n \leq 256$.

III. Secret-Shared Rotation

In this section, we provide a protocol for secret-shared rotation, the ideal functionality of which is shown in Figure 6.

Functionality $\mathcal{F}_{\mathsf{Rotate}}$

Input:
- $\mathcal{C}$: $\mathbf{x} \in \mathbb{Z}_{2^l}^n$
- $\mathcal{S}$: $s \in \mathbb{Z}_n$

Output:
- $\mathcal{C}$: $\langle \mathbf{x}' \rangle_{\mathcal{C}} \in \mathbb{Z}_{2^l}^n$
- $\mathcal{S}$: $\langle \mathbf{x}' \rangle_{\mathcal{S}} \in \mathbb{Z}_{2^l}^n$, s.t., $\mathbf{x}' = (\mathbf{x} \ggg s) = [x_{n-s}, x_{n-s+1}, ..., x_{n-1}, x_0, x_1, ..., x_{n-s-1}]$

Fig. 6: Ideal functionality for secret-shared rotation.

As rotation is special kind of permutation, we can directly employ the secret-shared permutation protocol described in Section II-D, with $\mathcal{S}$ using

$$\pi(i) = (i + s) \mod n$$

as the input permutation. However, this is an overkill given that rotation is much simpler than permutation.

3	4	0	1	2
3	4	0	1	2
3	4	0	1	2
3	4	0	1	2
3	4	0	1	2

Fig. 7: The matrix that replaces Figure 4(b) when rotation is applied with $s = 2$.

In Figure 7, a matrix is depicted as a replacement for Figure 4(b) when permutation is substituted with rotation. We use $s = 2$ as an example, i.e., each row was right-rotated for two positions. In the i-th row, the $(i+2)$-th position is punctured. Clearly, this matrix is more regular.

Recall that in secret-shared permutation, $\mathcal{S}$ uses n PPRFs to obtain the elements that are not punctured. This time, we aim to obtain all such elements with a *single* PPRF. To this end, we transform the matrix in Figure 7 into a rhombus shape that is shown in Figure 8. For each slant column of the rhombus, we have $\mathcal{C}$ uses a single seed to generate all elements in that column. Then, all the punctured elements are generated by the same seed. To this end, we could have $\mathcal{S}$ use a single PPRF to get all seeds except for the seed that generates the punctured elements.

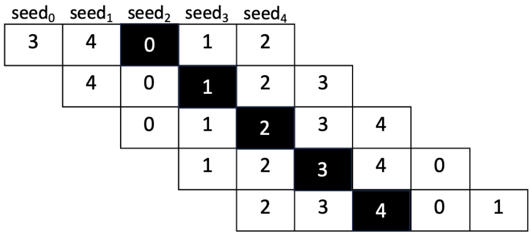


Fig. 8: Transformation of Figure 7 such that the punctured elements can be generated by a single seed.

$\mathcal{C}$ and $\mathcal{S}$ could locally expand the seeds to recover the matrix in Figure 7, and then calculate the row- and column-wise sums of this matrix. However, the row- and column-wise sums can be calculated even without recovering that matrix. Denote $\mathbf{d}_i$ as a column of elements generated by $\mathcal{C}$ from expanding the i-th seed. The row-wise sums can simply be calculated as $\bigoplus_i \mathbf{d}_i$,[1] as our transformation in Figure 8 does not change the elements in each row. To compute the column-wise sums of the matrix in Figure 7, $\mathcal{C}$ right-rotate each $\mathbf{d}_i$ by i elements, i.e., $\forall i \in \mathbb{Z}_n, \mathbf{d}'_i := (\mathbf{d}_i \ggg i)$. Then, the rhombus in Figure 8 becomes a new rhombus shown in Figure 9. The row-wise sums of this new rhombus (i.e., $\bigoplus_i \mathbf{d}'_i$) are exactly the column-wise sums of the matrix in Figure 7.

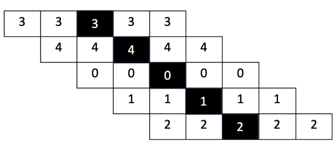


Fig. 9: Transformation of Figure 8 such that its row-wise sums are exactly the column-wise sums of the matrix in Figure 7.

The detailed protocol is shown in Figure 10.

Theorem 1. *The protocol in Figure 10 realizes the ideal functionality $\mathcal{F}_{\mathsf{Rotate}}$ in presence of a semi-honest adversary.*

Proof. (sketch)

Correctness. Notice that $c_i = r_i \oplus a_{i \ggg s}$ $\forall i \in \mathbb{Z}_n$. Then, $\mathcal{S}$'s output in Step 5 is:

$$\begin{aligned}((\mathbf{x} \oplus \mathbf{a}) \ggg s) \oplus \mathbf{c} &= (\mathbf{x} \ggg s) \oplus (\mathbf{a} \ggg s) \oplus \mathbf{c} \\ &= (\mathbf{x} \ggg s) \oplus (\mathbf{a} \ggg s) \oplus \mathbf{r} \oplus (\mathbf{a} \ggg s) \\ &= (\mathbf{x} \ggg s) \oplus \mathbf{r}.\end{aligned}$$

That means $\mathcal{S}$ and $\mathcal{C}$ secret-share $(\mathbf{x} \ggg s)$ at the end of the protocol.

[1]We use $\bigoplus$ to denote the element-wise addition between two vectors.

Protocol Π_{Rotate}

1) $\mathcal{C}$ and $\mathcal{S}$ run one execution of PPRF, where $\mathcal{S}$ uses s as its input. Let $\mathbf{seed}^{\mathcal{C}}$ and $\mathbf{seed}^{\mathcal{S}}$ be the output for $\mathcal{C}$ and $\mathcal{S}$ respectively.
2) $\mathcal{C}$ runs as follows:
   a) $\forall i \in \mathbb{Z}_n, \mathbf{d}_i \leftarrow \mathsf{PRG}(seed^{\mathcal{C}}_i)$, where $\mathbf{d}_i \in \mathbb{Z}^n_{2^l}$;
   b) $\mathbf{r} := \bigoplus_i \mathbf{d}_i$;
   c) $\forall i \in \mathbb{Z}_n, \mathbf{d}'_i := (\mathbf{d}_i \ggg i)$;
   d) $\mathbf{a} := \bigoplus_i \mathbf{d}'_i$;
3) $\mathcal{S}$ runs as follows:
   a) $\forall i \in \mathbb{Z}_n$ and $i \neq s, \mathbf{t}_i \leftarrow \mathsf{PRG}(seed^{\mathcal{S}}_i)$, where $\mathbf{t}_i \in \mathbb{Z}^n_{2^l}$;
   b) $\mathbf{t}_s := [0...0]$;
   c) $\mathbf{r}^* := \bigoplus_i \mathbf{t}_i$;
   d) $\forall i \in \mathbb{Z}_n, \mathbf{t}'_i := (\mathbf{t}_i \ggg i)$;
   e) $\mathbf{a}^* := \bigoplus_i \mathbf{t}'_i$;
   f) $\forall i \in \mathbb{Z}_n$, sets $c_i := r^*_i \oplus a^*_{i \ggg s}$.
4) $\mathcal{C}$ sends $\mathbf{x} \oplus \mathbf{a}$ to $\mathcal{S}$ and outputs $\mathbf{r}$.
5) $\mathcal{S}$ outputs $((\mathbf{x} \oplus \mathbf{a}) \ggg s) \oplus \mathbf{c}$.

Fig. 10: The secret-shared rotation protocol.

Security. We first consider the case where $\mathcal{C}$ is corrupt. We have $\mathcal{C}$ participate in two executions of the protocol, with $\mathcal{S}$ inputting s and s' respectively. Suppose $\mathcal{C}$ can tell the two executions apart. Then, we use $\mathcal{C}$ as a subroutine to build a distinguisher $\mathcal{A}$ that breaks the position hiding property of PPRF as follows:

1) $\mathcal{A}$ receives $(1^\lambda, n, i, i', \mathsf{view}^{\mathcal{C}}_{\mathsf{PPRF}})$, where $\mathsf{view}^{\mathcal{C}}_{\mathsf{PPRF}}$ contains $\mathbf{v}_{\mathcal{C}}$;
2) $\mathcal{A}$ runs Steps 2.a-2.d of Π_{Rotate} using $\mathbf{v}^{\mathcal{C}}$ as $\mathbf{seed}^{\mathcal{C}}$, and obtains $\mathbf{r}$ and $\mathbf{a}$;
3) $\mathcal{A}$ constructs $\mathsf{view}^{\mathcal{C}}_{\mathsf{Rotate}}$, which is $\mathsf{view}^{\mathcal{C}}_{\mathsf{PPRF}}$ augmented with $\mathbf{r}$ and $\mathbf{a}$;
4) $\mathcal{A}$ forwards $(1^\lambda, n, i, i', \mathsf{view}^{\mathcal{C}}_{\mathsf{Rotate}})$ to $\mathcal{C}$, treating (i, i') as (s, s');
5) $\mathcal{A}$ outputs what $\mathcal{C}$ outputs.

Then, if $\mathcal{C}$ can tell the two executions apart, $\mathcal{A}$ can break the position hiding property of PPRF.

Next, we consider the case where $\mathcal{S}$ is corrupt. We have $\mathcal{S}$ participate in two executions of the protocol, with $\mathcal{C}$ inputting $\mathbf{x}$ and $\mathbf{x}'$ respectively. We show the indistinguishability in a sequence of hybrids:

- $H_0 = (1^\lambda, n, \mathbf{x}, \mathsf{view}^{\mathcal{S}}_{\mathsf{Rotate}})$, where $\mathsf{view}^{\mathcal{S}}_{\mathsf{Rotate}}$ includes $\mathsf{view}^{\mathcal{S}}_{\mathsf{PPRF}}$ and $\mathbf{x} \oplus \mathbf{a}$.
- $H_1 = (1^\lambda, n, \mathbf{x}, \widetilde{\mathsf{view}}^{\mathcal{S}}_{\mathsf{Rotate}})$, where $\widetilde{\mathsf{view}}^{\mathcal{S}}_{\mathsf{Rotate}}$ is identical to $\mathsf{view}^{\mathcal{S}}_{\mathsf{Rotate}}$ except for replacing $\mathbf{a}$ with $\mathbf{a}' \in_\$ \mathbb{Z}^n_{2^l}$. By the value hiding property of PPRF, $\mathcal{C}$ could replace $seed^{\mathcal{C}}_s$ with a random seed from the domain. Then, by

the pseudorandom property of PRG, $\mathcal{C}$ could replace $\mathbf{d}_s$ with $\mathbf{d}'_s \in_\$ \mathbb{Z}_{2^l}^n$ in Step 2.a, which leads to a random $\mathbf{a}$ in Step 2.d. Therefore, $H_1 \approx H_0$.

- $H_2 = (1^\lambda, n, \mathbf{x}', \widetilde{\mathsf{view}}^{\mathcal{S}}_{\mathsf{Rotate}})$. As $\mathbf{a}'$ is an array of random numbers, $\mathbf{x} \oplus \mathbf{a}'$ and $\mathbf{x}' \oplus \mathbf{a}'$ are indistinguishable. Therefore, $H_2 \approx H_1$.

□

IV. Lookup Table with Arithmetic Shares

In this section, we present ROTL: our LUT protocol for arithmetic shares.

A. Strawman solution

The basic idea of ROTL is to have $\mathcal{C}$ and $\mathcal{S}$ jointly right-rotate the table $\mathbf{x}$ for $s \in_\$ \mathbb{Z}_n$ elements in the preprocessing phase, with both the rotated table and s being secret-shared between $\mathcal{C}$ and $\mathcal{S}$. Then, in the online phase, $\mathcal{C}$ and $\mathcal{S}$ can simply recover $i' := (i + s) \mod n$ and output the $(i+s)$-th element in the rotated table. To make $\mathcal{F}_{\mathsf{Rotate}}$ support a secret-shared input s, we first have $\mathcal{C}$ locally samples $\langle s \rangle_\mathcal{C} \in_\$ \mathbb{Z}_n$ and rotates $\mathbf{x}$ for $\langle s \rangle_\mathcal{C}$ elements:

$$\mathbf{y} := (\mathbf{x} \ggg \langle s \rangle_\mathcal{C}).$$

Then, we have $\mathcal{C}$ and $\mathcal{S}$ run

$$(\langle \mathbf{y}' \rangle_\mathcal{C}, \langle \mathbf{y}' \rangle_\mathcal{S}) \leftarrow \mathcal{F}_{\mathsf{Rotate}}(\mathbf{y}, \langle s \rangle_\mathcal{S}),$$

where $\langle s \rangle_\mathcal{S} \in_\$ \mathbb{Z}_n$ is sampled by $\mathcal{S}$.

This strawman solution is almost free in the online phase and only requires one call to $\mathcal{F}_{\mathsf{Rotate}}$ in the preprocessing phase. However, $\mathcal{C}$ has to input the whole table to $\mathcal{F}_{\mathsf{Rotate}}$. We aim to replace the table with a bit-vector to reduce the communication cost from $(2\lambda \log n + nl)$ bits to $(2\lambda \log n + n)$ bits.

B. ROTL

We have $\mathcal{C}$ replace $\mathbf{y} := (\mathbf{x} \ggg \langle s \rangle_\mathcal{C})$ with a bit-vector $\mathbf{b}$, where the $\langle s \rangle_\mathcal{C}$-th bit is 1 and other bits are 0s. Then, $\mathcal{C}$ and $\mathcal{S}$ run

$$(\langle \mathbf{b}' \rangle_\mathcal{C}, \langle \mathbf{b}' \rangle_\mathcal{S}) \leftarrow \mathcal{F}_{\mathsf{Rotate}}(\mathbf{b}, \langle s \rangle_\mathcal{S}).$$

Notice that the s-th bit in $\mathbf{b}'$ is 1 and other bits are 0s.

In the online phase, $\mathcal{C}$ and $\mathcal{S}$ recover $s' := (i - s) \mod n$. Then, they locally right-rotate $\langle \mathbf{b}' \rangle_\mathcal{C}$ and $\langle \mathbf{b}' \rangle_\mathcal{S}$ by s' elements, resulting in $\langle \mathbf{b}'' \rangle_\mathcal{C}$ and $\langle \mathbf{b}'' \rangle_\mathcal{S}$. The element to be chosen is the dot-product between $\mathbf{b}''$ and $\mathbf{x}$, the secret-shares of which can be computed locally by $\mathcal{C}$ and $\mathcal{S}$ given that $\mathbf{x}$ is public.

However, to compute the dot-product, the elements in $\mathbf{b}''$ need to be in the same domain (i.e., $\mathbb{Z}_{2^l}^n$) with the elements in $\mathbf{x}$. A naive solution is to use 2^l as the modulus for the elements in $\mathbf{b}$ at first hand, but this would negate the advantage of using a bit vector, resulting in the same complexity as the strawman solution. Instead, we maintain a modulus of 2 for the elements in $\mathbf{b}$ and aim to expand the modulus for the elements in $\mathbf{b}'$ from 2 to 2^l. To this end, we make the following observations:

- If $b'_i = 0$, the shares $(\langle b'_i \rangle^1_\mathcal{C}, \langle b'_i \rangle^1_\mathcal{S})$ can be either $(0,0)$ or $(1,1)$, meaning that

$$\langle b'_i \rangle^1_\mathcal{C} = \langle b'_i \rangle^1_\mathcal{S} \text{ when } b'_i = 0.$$

In this case, we could directly extend the modulus of b'_i from 2 to 2^l, with one party changing the sign of its share to produce $(\langle b'_i \rangle^l_\mathcal{C}, -\langle b'_i \rangle^l_\mathcal{S})$:

$$b'_i = \langle b'_i \rangle^l_\mathcal{C} + (-\langle b'_i \rangle^l_\mathcal{S}) \mod 2^l = 0.$$

- If $b'_i = 1$, the shares $(\langle b'_i \rangle^1_\mathcal{C}, \langle b'_i \rangle^1_\mathcal{S})$ can be either $(1,0)$ or $(0,1)$. The above procedure for case "$b'_i = 0$" may result in an error:
  - if $(\langle b'_i \rangle^1_\mathcal{C}, \langle b'_i \rangle^1_\mathcal{S}) = (1,0)$, $b'_i = \langle b'_i \rangle^l_\mathcal{C} + (-\langle b'_i \rangle^l_\mathcal{S}) \mod 2^l = 1$, which leads to a correct final output;
  - if $(\langle b'_i \rangle^1_\mathcal{C}, \langle b'_i \rangle^1_\mathcal{S}) = (0,1)$, $b'_i = \langle b'_i \rangle^l_\mathcal{C} + (-\langle b'_i \rangle^l_\mathcal{S}) \mod 2^l = -1$, which flips the sign of the final output.

To get rid of this error, we have $\mathcal{C}$ and $\mathcal{S}$ locally compute $\langle sum \rangle^l_\mathcal{C} := \sum_j \langle b'_j \rangle^l_\mathcal{C}$ and $\langle sum \rangle^l_\mathcal{S} := \sum_j (-\langle b'_j \rangle^l_\mathcal{C})$ respectively:
  - if $(\langle b'_i \rangle^1_\mathcal{C}, \langle b'_i \rangle^1_\mathcal{S}) = (1,0)$, $sum = 1$;
  - if $(\langle b'_i \rangle^1_\mathcal{C}, \langle b'_i \rangle^1_\mathcal{S}) = (0,1)$, $sum = -1$.

Then, we only need to multiply sum to the dot-product. The detailed protocol is shown in Figure 11.

Protocol Π_{LUT}

Preprocessing:

1) $\mathcal{C}$ locally samples $\langle s \rangle_\mathcal{C} \in_\$ \mathbb{Z}_n$ and generates a bit-vector $\mathbf{b}$, where the $\langle s \rangle_\mathcal{C}$-th bit is 1 and other bits are 0s.
2) $\mathcal{S}$ locally samples $\langle s \rangle_\mathcal{S} \in_\$ \mathbb{Z}_n$.
3) $\mathcal{C}$ and $\mathcal{S}$ run $(\langle \mathbf{b}' \rangle_\mathcal{C}, \langle \mathbf{b}' \rangle_\mathcal{S}) \leftarrow \mathcal{F}_{\mathsf{Rotate}}(\mathbf{b}, \langle s \rangle_\mathcal{S})$.
4) $\mathcal{C}$ locally computes $\langle sum \rangle^l_\mathcal{C} := \sum_j \langle b'_j \rangle^l_\mathcal{C}$.
5) $\mathcal{S}$ locally computes $\langle sum \rangle^l_\mathcal{S} := \sum_j -\langle b'_j \rangle^l_\mathcal{C}$.

Online:

1) $\mathcal{C}$ and $\mathcal{S}$ recover $s' := (i - s)$.
2) $\mathcal{C}$ locally right-rotates $\langle \mathbf{b}' \rangle_\mathcal{C}$ by s' elements, resulting in $\langle \mathbf{b}'' \rangle_\mathcal{C}$, and computes $\langle z \rangle^l_\mathcal{C} := \sum_j (\langle b''_j \rangle^l_\mathcal{C} \cdot x_j)$.
3) $\mathcal{S}$ locally right-rotates $\langle \mathbf{b}' \rangle_\mathcal{S}$ by s' elements, resulting in $\langle \mathbf{b}'' \rangle_\mathcal{S}$, and computes $\langle z \rangle^l_\mathcal{S} := \sum_j (-\langle b''_j \rangle^l_\mathcal{S} \cdot x_j)$.
4) $\mathcal{C}$ and $\mathcal{S}$ run $(\langle z' \rangle^l_\mathcal{C}, \langle z' \rangle^l_\mathcal{S}) \leftarrow \mathcal{F}_{\mathsf{Mult}}(sum, z)$.
5) $\mathcal{C}$ outputs $\langle z' \rangle^l_\mathcal{C}$ and $\mathcal{S}$ outputs $\langle z' \rangle^l_\mathcal{S}$.

Fig. 11: The ROTL protocol.

Theorem 2. *The protocol in Figure 11 realizes the ideal functionality $\mathcal{F}_{\mathsf{LUT}}$ in presence of a semi-honest adversary.*

Proof. (sketch)

Correctness. After right-rotating $\mathbf{b}'$ by s', $\mathcal{C}$ and $\mathcal{S}$ get a secret-shared bit vector $\mathbf{b}''$, with $b''_i = 1$ and $b''_{j \neq i} = 0$. After computing the doc-product, $z = x_i \cdot sum$ with $sum = 1$ or -1. Then, $z' = x_i$.

Security. Clearly, all computations are local except one call to $\mathcal{F}_{\mathsf{Rotate}}$ and one call to $\mathcal{F}_{\mathsf{Mult}}$. Therefore, the security of Π_{LUT} is directly implied by the security of Π_{Rotate} and Π_{Mult}. $\square$

An optimization is that, instead of having $\mathcal{S}$ locally sample $\langle s\rangle_{\mathcal{S}} \in_{\$} \mathbb{Z}_n$ at Step 2 (in the preprocessing phase), we could rely on PPRF to "sample" $\langle s\rangle_{\mathcal{S}}$. Namely, we replace OTs with random-OTs in PPRF, resulting in a random punctured position, which could be treated as $\langle s\rangle_{\mathcal{S}}$. As a result, we reduce the communication of Π_{Rotate} from $(2\lambda \log n + n)$ bits to $((0.6+\lambda)\log n + n)$ bits.

Another optimization is to replace $\mathcal{F}_{\mathsf{Mult}}$ with $\mathcal{F}_{\mathsf{MUX}}$ at Step 4 (in the online phase), as $\mathcal{F}_{\mathsf{MUX}}$ can be realized with two executions of 1-out-of-2 OT, much cheaper than $\mathcal{F}_{\mathsf{Mult}}$. The fundamental idea of this optimization is:

$$z' \leftarrow \mathcal{F}_{\mathsf{MUX}}(2z, \beta) - z,$$

where β is a bit indicating the sign of sum, i.e., $\beta \leftarrow \mathcal{F}_{\mathsf{CMP}}(sum, 0)$.

Given that $sum = 1$ or -1, we could even get β for free (without invoking $\mathcal{F}_{\mathsf{CMP}}$). This is based on the observation that β is equal to the inverse of any but the least-significant bit (LSB) of sum, (notice that $\mathsf{LSB}(sum)$ is always 1 no matter whether $sum = 1$ or -1). Let $\alpha_i^{\mathcal{C}}$ be the i-th bit of $\langle sum\rangle_{\mathcal{C}}^{l}$ and $\alpha_i^{\mathcal{S}}$ be the i-th bit of $\langle sum\rangle_{\mathcal{S}}^{l}$, with $i = 0$ denoting the least-significant bit. Then,

$$\beta = 1 \oplus \alpha_i^{\mathcal{C}} \oplus \alpha_i^{\mathcal{S}} \oplus c_{i-1} \ \forall\ i > 0,$$

where c_{i-1} is the carry-bit from $i-1$. Given that $c_0 = 0$, we could directly use

$$\langle\beta\rangle_{\mathcal{C}} := 1 \oplus \alpha_1^{\mathcal{C}}, \quad \langle\beta\rangle_{\mathcal{S}} := \alpha_1^{\mathcal{S}}$$

as the input shares of β for $\mathcal{F}_{\mathsf{MUX}}$.

C. Comparison

Table II provides a theoretical comparison between ROTL and existing LUT protocols that support arithmetic shares.

In fact, OTTT [19] and FLUTE [4] respectively involve a communication of $(|\mathsf{MT}|+4)\cdot(\log n - 1)nl$ bits and $(|\mathsf{MT}|+4)\cdot(n - \log n - 1)$ bits during preprocessing, where $|\mathsf{MT}|$ denotes the communication cost for generating a boolean multiplication triplet. Notice that we assume a different $|\mathsf{MT}|$ with those in [9] and [4]:

- [9] assumed a relatively large $|\mathsf{MT}|$ (i.e., 138 bits), as they use IKNP [18], [1] for oblivious transfer extensions.
- [4] replaced IKNP with silent OT extension [3], which reduces $|\mathsf{MT}|$ to 0.236 bits, but requires more computation.
- We use Ferret-OT [31], which achieves a better trade-off between communication and computation. The communication cost per random-OT is 0.6 bits and $|\mathsf{MT}|$ is 1.2 bits.

Table III summarizes the improvement of ROTL over SP-LUT in terms of total communication (ROTL is clearly better than OTTT and OP-LUT, hence we focus on comparing with SP-LUT).

In terms of computation, ROTL requires $\log n \cdot \mathsf{ROT} + (n + n^2/\lambda)\mathsf{AES} + 3n \cdot \mathsf{XOR}$ during preprocessing and $n \cdot \mathsf{XOR} + 1\mathsf{Mult}$ during online computation:

- It requires $n \cdot \mathsf{AES}$ for PPRF and $n^2/\lambda \cdot \mathsf{AES}$ for generating the rhombus of Figure 8.
- Recall that $n \le 256$, hence we can put the bits in each $\mathbf{d}_i$ (same for $\mathbf{d}'_i$, $\mathbf{t}_i$, $\mathbf{t}'_i$) in Figure 10 into a single uint256 and computes $n \cdot \mathsf{XOR}$ with a single CPU instruction (same as one XOR). Consequently, it requires $2n \cdot \mathsf{XOR} + n \cdot \mathsf{Shift}$ to compute the row- and column-wise sums of the rhombus. Given that shifting an uint256 also requires a single CPU instruction, it requires $3n \cdot \mathsf{XOR}$ in total.
- During online computation, it requires n plaintext multiplications to compute a dot-product, which is equivalent to $2n \cdot \mathsf{XOR}$, as a plaintext multiplication/addition also requires a single CPU instruction. Additionally, it requires a MUX operation, which equals to OTs.

We refer to Section VII for a detailed empirical comparison.

V. Lookup Table with Boolean Inputs

Arguably, ROTL achieves the best tradeoff between computation and communication, but it necessitates augmentation with A2B conversions to support boolean inputs. Although boolean inputs are less commonly employed for LUTs, they find application in the evaluation of boolean circuits. For completeness, we introduce another LUT solution named FLUTE+ for boolean inputs. It can be considered as an optimization of FLUTE [4], with a reduction of the computational complexity from $O(l \cdot n^2)$ to $O(n \log n + l \cdot n)$, and a shift of the $O(n \log n)$ computation to the preprocessing phase.

The fundamental idea of FLUTE lies in the conversion of LUT description into boolean expressions. For example, let $\log n = 2$, x_0, x_1 be the two input bits, and $y_1 \cdots y_l$ be the l output bits, the lookup table is:

x_0 x_1	$y_1 \cdots y_l$
0 0	$a_1 \cdots a_l$
0 1	$b_1 \cdots b_l$
1 0	$c_1 \cdots c_l$
1 1	$d_1 \cdots d_l$

They represent each output bit as:

$$y_i = (\overline{x}_0 \wedge \overline{x}_1 \wedge a_i) \oplus (x_0 \wedge \overline{x}_1 \wedge b_i) \oplus (\overline{x}_0 \wedge x_1 \wedge c_i) \oplus (x_0 \wedge x_1 \wedge d_i).$$

In the preprocessing phase, $\mathcal{C}$ and $\mathcal{S}$ generate secret-shares of two random bits α_0 and α_1, and use $\mathcal{F}_{\mathsf{AND}}$ to compute the secret-share of $\beta := \alpha_0 \wedge \alpha_1$. In the online phase, they reveal

TABLE II: Comparison with existing LUT protocols that support arithmetic inputs. A table has n l-bit elements. We use Ferret-OT [31] for OT instances, which roughly requires 0.6 bits per random-OT. When calculating the computational overhead, we only consider one party, as the two parties can run in parallel.

Protocol	Communication (bits)		Computation	
	preprocessing	online	preprocessing	online
OTTT [19]	$5.2(\log n - 1)nl$	$2\log n$	$(\log n - 1)nl \cdot \mathsf{ROT} + nl \cdot \mathsf{XOR}$	$1\mathsf{XOR}$
OP-LUT [9]	$n^2 l - 0.4\log n$	$2\log n$	$\log n \cdot \mathsf{ROT} + n^2 \log n \cdot \mathsf{XOR}$	$1\mathsf{XOR}$
SP-LUT [9]	$0.6\log n$	$nl + \log n$	$\log n \cdot \mathsf{ROT} + n\log n \cdot \mathsf{XOR}$	$n \cdot \mathsf{XOR}$
ROTL	$(0.6+\lambda)\log n + n$	$2\log n$	$\log n \cdot \mathsf{ROT} + (n + n^2/\lambda)\mathsf{AES} + 3n \cdot \mathsf{XOR}$	$2n \cdot \mathsf{XOR} + 1\mathsf{MUX}$

TABLE III: Improvement factor of total communication of ROTL over SP-LUT.

n \ l	8	16	32	64	128
4	0.12	0.20	0.33	0.50	0.66
8	0.16	0.29	0.50	0.79	1.13
16	0.24	0.45	0.79	1.32	1.98
32	0.38	0.72	1.32	2.25	3.51
64	0.64	1.21	2.25	3.94	6.31
128	1.09	2.10	3.93	6.99	11.5
256	1.91	3.69	6.98	12.6	21.0

$m_0 := \alpha_0 \oplus x_0$ and $m_1 := \alpha_1 \oplus x_1$. Then, for example,

$$\begin{aligned}
&\overline{x}_0 \wedge \overline{x}_1 \wedge a_i \\
&= (\overline{m}_0 \oplus \alpha_0) \wedge (\overline{m}_1 \oplus \alpha_1) \wedge a_i \\
&= (\overline{m}_0 \wedge \overline{m}_1 \oplus \overline{m}_0 \wedge \alpha_1 \oplus \overline{m}_1 \wedge \alpha_0 \oplus \alpha_0\alpha_1) \wedge a_i \\
&= (\overline{m}_0 \wedge \overline{m}_1 \oplus \overline{m}_0 \wedge \langle\alpha_1\rangle_{\mathcal{C}} \oplus \overline{m}_1 \wedge \langle\alpha_0\rangle_{\mathcal{C}} \oplus \langle\beta\rangle_{\mathcal{C}}) \wedge a_i \\
&\qquad \oplus (0 \oplus \overline{m}_0 \wedge \langle\alpha_1\rangle_{\mathcal{S}} \oplus \overline{m}_1 \wedge \langle\alpha_0\rangle_{\mathcal{S}} \oplus \langle\beta\rangle_{\mathcal{S}}) \wedge a_i,
\end{aligned}$$

the secret-share of which can be computed locally by $\mathcal{S}$ and $\mathcal{C}$. For the public values such as $\overline{m}_0 \wedge \overline{m}_1$, $\mathcal{C}$ holds $\overline{m}_0 \wedge \overline{m}_1$ and $\mathcal{S}$ holds 0. FLUTE [4] runs the above process for all n possible inputs ($n = 4$ in our example) and l output bits, which involves $O(l \cdot n^2)$ online computation in total.

Our first observation reveals that $(\overline{x}_0 \wedge \overline{x}_1,\ x_0 \wedge \overline{x}_1,\ \overline{x}_0 \wedge x_1,\ x_0 \wedge x_1)$ are consistent across all output bits, hence we can compute them once and reuse the results uniformly. This optimization reduces the online computation of FLUTE from $O(l \cdot n^2)$ to $O(n^2 + l \cdot n)$. We further observe that the online computation for each party (e.g., $\mathcal{C}$) to compute an output bit is:

$$\begin{aligned}
&(\overline{m}_0 \wedge \overline{m}_1 \oplus \overline{m}_0 \wedge \langle\alpha_1\rangle_{\mathcal{C}} \oplus \overline{m}_1 \wedge \langle\alpha_0\rangle_{\mathcal{C}} \oplus \langle\beta\rangle_{\mathcal{C}}) \wedge a_i \oplus \\
&(m_0 \wedge \overline{m}_1 \oplus m_0 \wedge \langle\alpha_1\rangle_{\mathcal{C}} \oplus \overline{m}_1 \wedge \langle\alpha_0\rangle_{\mathcal{C}} \oplus \langle\beta\rangle_{\mathcal{C}}) \wedge b_i \oplus \\
&(\overline{m}_0 \wedge m_1 \oplus \overline{m}_0 \wedge \langle\alpha_1\rangle_{\mathcal{C}} \oplus m_1 \wedge \langle\alpha_0\rangle_{\mathcal{C}} \oplus \langle\beta\rangle_{\mathcal{C}}) \wedge c_i \oplus \\
&(m_0 \wedge m_1 \oplus m_0 \wedge \langle\alpha_1\rangle_{\mathcal{C}} \oplus m_1 \wedge \langle\alpha_0\rangle_{\mathcal{C}} \oplus \langle\beta\rangle_{\mathcal{C}}) \wedge d_i.
\end{aligned}$$

Although $\mathcal{C}$ knows m_0 and m_1 only in the online phase, it could enumerate the possible values of m_0 and m_1 in the preprocessing phase and compute:

$$\begin{aligned}
\langle s_0\rangle_{\mathcal{C}} &:= 0 \wedge 0 \oplus 0 \wedge \langle\alpha_1\rangle_{\mathcal{C}} \oplus 0 \wedge \langle\alpha_0\rangle_{\mathcal{C}} \oplus \langle\beta\rangle_{\mathcal{C}}, \\
\langle s_1\rangle_{\mathcal{C}} &:= 0 \wedge 1 \oplus 0 \wedge \langle\alpha_1\rangle_{\mathcal{C}} \oplus 1 \wedge \langle\alpha_0\rangle_{\mathcal{C}} \oplus \langle\beta\rangle_{\mathcal{C}}, \\
\langle s_2\rangle_{\mathcal{C}} &:= 1 \wedge 0 \oplus 1 \wedge \langle\alpha_1\rangle_{\mathcal{C}} \oplus 0 \wedge \langle\alpha_0\rangle_{\mathcal{C}} \oplus \langle\beta\rangle_{\mathcal{C}}, \\
\langle s_3\rangle_{\mathcal{C}} &:= 1 \wedge 1 \oplus 1 \wedge \langle\alpha_1\rangle_{\mathcal{C}} \oplus 1 \wedge \langle\alpha_0\rangle_{\mathcal{C}} \oplus \langle\beta\rangle_{\mathcal{C}}.
\end{aligned}$$

After knowing m_0 and m_1 in the online phase, they could locally re-arrange the order of $(s_0,\ s_1,\ s_2,\ s_3)$. This transition shifts the $O(n^2)$ computation to the preprocessing phase, leaving only $O(l \cdot n)$ local computation online.

We further reduce the preprocessing computation from $O(n^2)$ to $O(n \log n)$ by leveraging the *butterfly diagram optimization* [6]. In more detail, we transform $(s_0,\ s_1,\ s_2,\ s_3)$ to:

$$\begin{aligned}
\langle s_0\rangle_{\mathcal{C}} &= (0 \oplus \langle\alpha_1\rangle_{\mathcal{C}}) \wedge (0 \oplus \langle\alpha_0\rangle_{\mathcal{C}}) \\
&= \langle\alpha_1\rangle_{\mathcal{C}} \wedge \langle\alpha_0\rangle_{\mathcal{C}}, \\
\langle s_1\rangle_{\mathcal{C}} &= (0 \oplus \langle\alpha_1\rangle_{\mathcal{C}}) \wedge (1 \oplus \langle\alpha_0\rangle_{\mathcal{C}}) \\
&= \langle\alpha_1\rangle_{\mathcal{C}} \wedge (1 \oplus \langle\alpha_0\rangle_{\mathcal{C}}), \\
&= \langle\alpha_1\rangle_{\mathcal{C}} \oplus \langle\alpha_1\rangle_{\mathcal{C}} \wedge \langle\alpha_0\rangle_{\mathcal{C}}, \\
\langle s_2\rangle_{\mathcal{C}} &= (1 \oplus \langle\alpha_1\rangle_{\mathcal{C}}) \wedge (0 \oplus \langle\alpha_0\rangle_{\mathcal{C}}), \\
&= (1 \oplus \langle\alpha_1\rangle_{\mathcal{C}}) \wedge \langle\alpha_0\rangle_{\mathcal{C}}, \\
&= \langle\alpha_0\rangle_{\mathcal{C}} \oplus \langle\alpha_1\rangle_{\mathcal{C}} \wedge \langle\alpha_0\rangle_{\mathcal{C}}, \\
\langle s_3\rangle_{\mathcal{C}} &= (1 \oplus \langle\alpha_1\rangle_{\mathcal{C}}) \wedge (1 \oplus \langle\alpha_0\rangle_{\mathcal{C}}), \\
&= 1 \oplus \langle\alpha_0\rangle_{\mathcal{C}} \oplus \langle\alpha_1\rangle_{\mathcal{C}} \oplus \langle\alpha_1\rangle_{\mathcal{C}} \wedge \langle\alpha_0\rangle_{\mathcal{C}}.
\end{aligned}$$

To compute $(\langle s_0\rangle_{\mathcal{C}}, \langle s_1\rangle_{\mathcal{C}}, \langle s_2\rangle_{\mathcal{C}}, \langle s_3\rangle_{\mathcal{C}})$ efficiently, $\mathcal{C}$ first sets:

$$\begin{aligned}
\langle s_0\rangle_{\mathcal{C}} &:= \langle\alpha_0\rangle_{\mathcal{C}} \wedge \langle\alpha_1\rangle_{\mathcal{C}}, \\
\langle s_1\rangle_{\mathcal{C}} &:= \langle\alpha_1\rangle_{\mathcal{C}}, \\
\langle s_2\rangle_{\mathcal{C}} &:= \langle\alpha_0\rangle_{\mathcal{C}}, \\
\langle s_3\rangle_{\mathcal{C}} &:= 1.
\end{aligned}$$

TABLE IV: Comparison with FLUTE. A table has n l-bit elements. We use Ferret-OT [31] for OT instances, which roughly requires 0.6 bits per random-OT. When calculating the computational overhead, we only consider one party, as the two parties can run in parallel.

Protocol	Communication (bits)		Computation	
	preprocessing	online	preprocessing	online
FLUTE [4]	$5.2(n - \log n - 1)$	$2\log n$	$(n - \log n - 1)(\mathsf{ROT} + \mathsf{XOR})$	$(ln^2 + ln)\mathsf{XOR}$
FLUTE+	$5.2(n - \log n - 1)$	$2\log n$	$(n - \log n - 1)(\mathsf{ROT} + \mathsf{XOR}) + n\log n\mathsf{XOR}$	$(ln + n\log n)\mathsf{XOR}$

Then, $\mathcal{C}$ computes:

$$\begin{aligned}\langle s_3\rangle_{\mathcal{C}} &:= \langle s_3\rangle_{\mathcal{C}} \oplus \langle s_2\rangle_{\mathcal{C}},\\ &= 1 \oplus \langle \alpha_0\rangle_{\mathcal{C}}.\\ \langle s_1\rangle_{\mathcal{C}} &:= \langle s_1\rangle_{\mathcal{C}} \oplus \langle s_0\rangle_{\mathcal{C}},\\ &= \langle \alpha_1\rangle_{\mathcal{C}} \oplus \langle \alpha_0\rangle_{\mathcal{C}} \wedge \langle \alpha_1\rangle_{\mathcal{C}}\end{aligned}$$

Next, $\mathcal{C}$ computes:

$$\begin{aligned}\langle s_2\rangle_{\mathcal{C}} &:= \langle s_2\rangle_{\mathcal{C}} \oplus \langle s_0\rangle_{\mathcal{C}},\\ &= \langle \alpha_0\rangle_{\mathcal{C}} \oplus \langle \alpha_1\rangle_{\mathcal{C}} \wedge \langle \alpha_0\rangle_{\mathcal{C}},\\ \langle s_3\rangle_{\mathcal{C}} &:= \langle s_3\rangle_{\mathcal{C}} \oplus \langle s_1\rangle_{\mathcal{C}},\\ &= 1 \oplus \langle \alpha_0\rangle_{\mathcal{C}} \oplus \langle \alpha_1\rangle_{\mathcal{C}} \oplus \langle \alpha_0\rangle_{\mathcal{C}} \wedge \langle \alpha_1\rangle_{\mathcal{C}}.\end{aligned}$$

For $\log n$ input bits, the process for generating ($\langle s_0\rangle_{\mathcal{C}}$, $\langle s_1\rangle_{\mathcal{C}}$, $\langle s_2\rangle_{\mathcal{C}}$, $\cdots \langle s_{n-1}\rangle_{\mathcal{C}}$) is as follows:

1) $\forall\ i \in [0, n-1]$: $Q_i \leftarrow \{j | j\text{-th bit of } i \text{ is } 0\}$
   - if $Q_i = \varnothing$, $\langle s_i\rangle_{\mathcal{C}} := 1$;
   - otherwise, $\langle s_i\rangle_{\mathcal{C}} := \wedge_{j\in Q_i} \langle \alpha_j\rangle_{\mathcal{C}}$.

   This step requires $(n - \log n - 1)$ invocations of $\mathcal{F}_{\mathsf{AND}}$.
2) $\forall\ j \in [0, \log n - 1]$ and $\forall\ i \in [0, n-1]$: if the j-th bit of i is 1, $\langle s_i\rangle_{\mathcal{C}} := \langle s_i\rangle_{\mathcal{C}} \oplus \langle s_{i+2^j}\rangle_{\mathcal{C}}$.

$\mathcal{S}$ runs symmetrically as $\mathcal{C}$, except that, in Step 1, $\mathcal{S}$ sets $\langle s_i\rangle_{\mathcal{S}} := 0$ when $Q_i = \varnothing$ to have $s_i = 1$.

This optimization effectively reduces the preprocessing computation from $O(n^2)$ to $O(n \log n)$.

Table IV shows the theoretical comparison between FLUTE and FLUTE+. Their communication costs are identical as we focus on optimizing computation. In the online phase, FLUTE+ involves a slight additional effort in re-arranging the order of (s_0, s_1, s_2, s_3), amounting to approximately $n \log n$ XOR operations.

VI. Secure Comparison

In this section, we provide a protocol for secure comparison. Recall that the ideal functionality of secure comparison is:

$$b \leftarrow \mathsf{CMP}(x, y)\text{: } b = 1 \text{ if } x \geq y,\ b = 0 \text{ otherwise;}$$

where x and y are two signed integers in $\mathbb{Z}_{2^l}$. To compute a secret-shared b, we could have $\mathcal{S}$ and $\mathcal{C}$ compute an arithmetic share of $a := x - y$. Then, b is the most-significant bit (MSB) of a:

$$b = 1 \oplus \mathsf{MSB}(a).$$

Let $\langle a\rangle_{\mathcal{C}} = msb_{\mathcal{C}} || \langle a'\rangle_{\mathcal{C}}$ and $\langle a\rangle_{\mathcal{S}} = msb_{\mathcal{S}} || \langle a'\rangle_{\mathcal{S}}$, then

$$\mathsf{MSB}(a) = msb_{\mathcal{C}} \oplus msb_{\mathcal{S}} \oplus carry,$$

where $carry = \mathbf{1}\{\langle a'\rangle_{\mathcal{C}} + \langle a'\rangle_{\mathcal{S}} \geq 2^{l-1}\}$. If $\mathcal{S}$ and $\mathcal{C}$ can compute the secret-share of $carry$, they can obtain the secret-share of $\mathsf{MSB}(a)$.

Let $c = \langle a'\rangle_{\mathcal{C}}$ and $d = 2^{l-1} - \langle a'\rangle_{\mathcal{S}}$, then

$$carry = \mathbf{1}\{c \geq d\} = 1 \oplus \mathbf{1}\{c < d\},$$

where c and d are two unsigned integers in $\mathbb{Z}_{2^{l-1}}$. Notice that the computation of $\mathbf{1}\{c < d\}$ is a *millionaires' problem*, the ideal functionality of which is shown in Figure 12.

Functionality $\mathcal{F}_{\mathsf{MILL}}$

Input:
- $\mathcal{C}$: $c \in \mathbb{Z}_{2^{l-1}}$
- $\mathcal{S}$: $d \in \mathbb{Z}_{2^{l-1}}$

Output:
- $\mathcal{C}$: $\langle \beta\rangle_{\mathcal{C}} \in \mathbb{Z}_2$
- $\mathcal{S}$: $\langle \beta\rangle_{\mathcal{S}} \in \mathbb{Z}_2$, s.t., $\beta = \mathbf{1}\{c < d\}$

Fig. 12: Ideal functionality for secure comparison.

The insight of our millionaires' protocol is to have $\mathcal{S}$ and $\mathcal{C}$ compute: $e = c - d \mod 2^l$, and (arithmetically) secret-share the result. Then, they invoke $\mathcal{F}_{\mathsf{LUT}}(\langle e\rangle_{\mathcal{C}}, \langle e\rangle_{\mathcal{S}})$ with a public table:

$$[\underbrace{0, ..., 0}_{2^{l-1}}, \underbrace{1, ..., 1}_{2^{l-1}}].$$

It is noteworthy that the modular for e is 2^l (instead of 2^{l-1}), because c and d are unsigned integers. [2]

With ROTL (cf. Section IV), all the expensive operations are shifted to the preprocessing phase, leaving merely one $\mathcal{F}_{\mathsf{MUX}}$ in the online phase. However, this is feasible only when $n = 2^l$ is not enough. Recall that ROTL necessitates $O(n)$ communication and $O(n^2)$ computation in the preprocessing phase. To this end, we reduce the size of LUT by partitioning the inputs into smaller blocks. Figure 13 shows the details of our millionaires' protocol. Its security is straightforward as

[2]If the modular is 2^{l-1}, the table would be $\mathbf{t} = [\underbrace{1, ..., 1}_{2^{l-2}}, \underbrace{0, ..., 0}_{2^{l-2}}]$. Take $c = 2^{l-2} + 2$ and $d = 1$ as an example, $e = 2^{l-2} + 1$ should return 1, but it will return 0.

all operations are local except some invocations of $\mathcal{F}_{\mathsf{LUT}}$ and $\mathcal{F}_{\mathsf{AND}}$.

Protocol Π_{MILL}

1) $\mathcal{C}$ and $\mathcal{S}$ parse c and d as smaller blocks of l' bits, with $m = \lceil \frac{l-1}{l'} \rceil$:
$$c = c_{m-1}||...||c_0,$$
$$d = d_{m-1}||...||d_0.$$
2) For $i = 0$ to $m-1$, $\mathcal{C}$ and $\mathcal{S}$ compute $e_i = c_i - d_i \mod 2^{m+1}$, and invoke $\mathcal{F}_{\mathsf{LUT}}(\langle e_i \rangle_{\mathcal{C}}, \langle e_i \rangle_{\mathcal{S}})$ with two public tables:
$$[\underbrace{0,...,0}_{2^m}, \underbrace{1,...,1}_{2^m}], \quad [1, \underbrace{0,...,0}_{2^{m+1}-1}]$$
and get the secret-shares of
$\mathtt{lt}_i = \mathbf{1}\{c_i < d_i\}$ and $\mathtt{eq}_i = \mathbf{1}\{c_i = d_i\}$
respectively. Notice that, since the inputs to both tables are identical (i.e., e_i), a single ROTL invocation is enough.
3) $\mathcal{C}$ and $\mathcal{S}$ output:
$$\mathtt{lt}_{m-1} \oplus (\mathtt{eq}_{m-1} \wedge \mathtt{lt}_{m-2}) \oplus$$
$$(\mathtt{eq}_{m-1} \wedge \mathtt{eq}_{m-2} \wedge \mathtt{lt}_{m-3}) \oplus ... \oplus$$
$$(\mathtt{eq}_{m-1} \wedge ... \wedge \mathtt{eq}_1 \wedge \mathtt{lt}_0)$$

Fig. 13: The millionaires' protocol.

VII. Evaluation

In this section, we empirically compare ROTL and FLUTE+ with existing LUT protocols. We also compare our secure comparison protocol with the state-of-the-art. We also implement Softmax and GELU based on ROTL, and systematically evaluate them with the real parameters in GPT-2 [25].

A. Implementation

We fully implemented ROTL and FLUTE+ in C++. We use AES for length-doubling PRG [11] and use Ferret-OT [31] from EMP-toolkit[3] for OT instances. For PPRF, we incorporated the half-tree [12] optimization to reduce both communication and computation. We use AXV2 (Advanced Vector Extensions) to accelerate the operations for 256-bit-width integers (i.e., uint256).

We also re-implemented FLUTE [4] in C++, for two reasons: 1) the original FLUTE implementation[4] uses Silver [7] for OT instances, which has been proved to be insecure [26], and we replaced it with Ferret-OT; 2) to have a fair comparison, we adopt the same library (i.e., EMP-toolkit) for the cryptographic operations used in all LUT protocols to be benchmarked.

We set the security parameter λ as 128 for all implementations.

[3]https://github.com/emp-toolkit/emp-ot

[4]https://github.com/encryptogroup/FLUTE

B. Experimental setup

We consider both LAN and WAN in our benchmark: in LAN, the bandwidth is 3000 Mbps and RTT is 0.8ms; in WAN, the bandwidth is 100 Mbps and RTT is 50ms. All experiments were performed on AWS c5.9xlarge instances with Intel Xeon 8000 series CPUs at 3.6GHz, and they were conducted using a single thread.

Since FLUTE has been proved to be better than OTTT and OP-LUT [4] in almost all aspects, we omit OTTT and OP-LUT in our benchmarks and focus on comparing with FLUTE and SP-LUT.

If we measure the performance of a single LUT instance, the error could be substantial as a single LUT instance runs in μs. To avoid this, we sequentially run 100 000 LUT instances and report the average communication and runtime for all benchmarks.

C. LUT evaluation

We initially configure the output bit-length l to be 64 and evaluate the LUT protocols across various table lengths n.

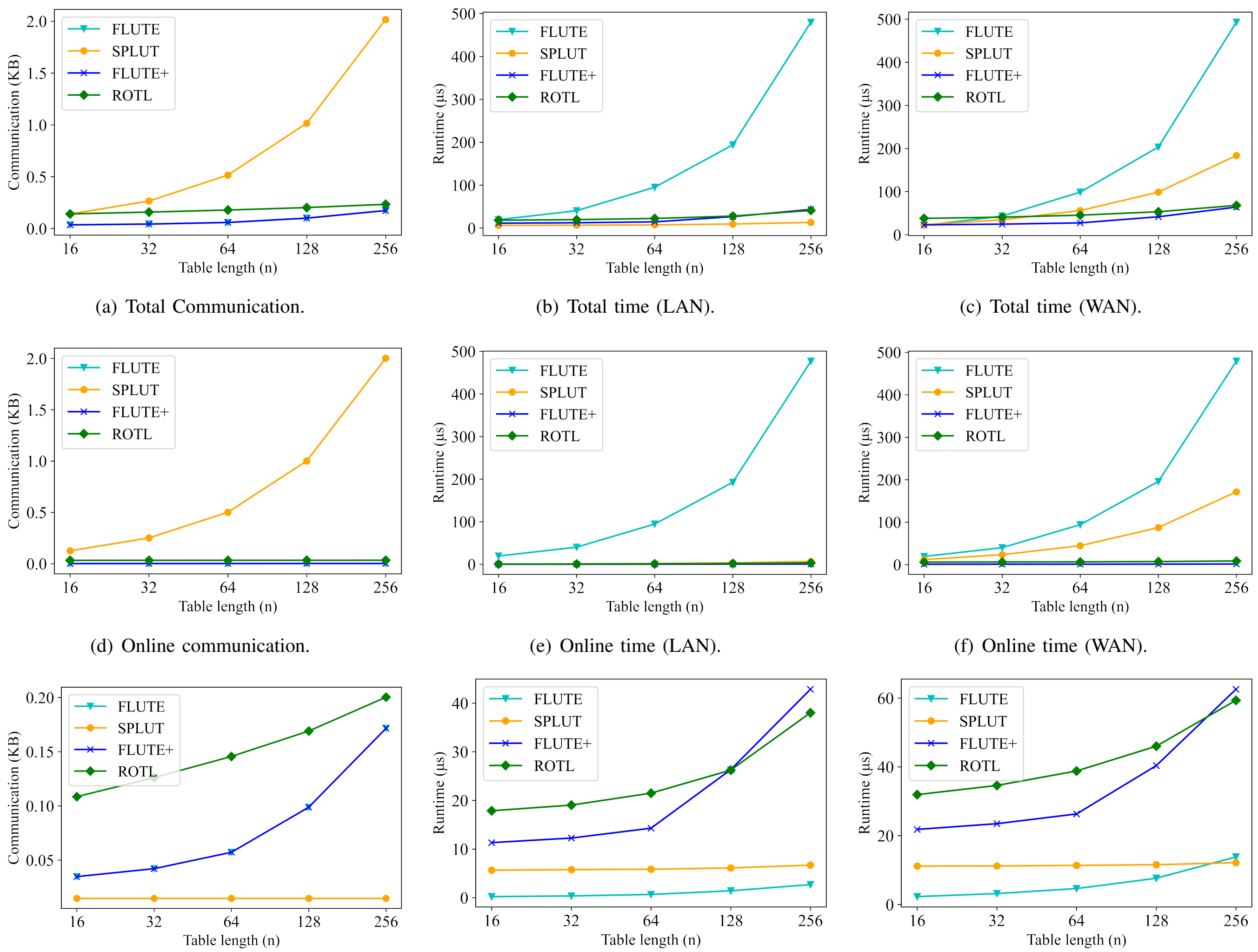
In Figure 14(a), the comparison is presented in terms of total communication. SP-LUT exhibits the poorest performance in this regard as it incurs $O(l \cdot n)$ communication, while the others operate at $O(n)$. Consequently, when $n = 256$, SP-LUT is approximately $11.6\times$ expensive than others. ROTL's communication is roughly $2\times$ that of FLUTE and FLUTE+ when n is small, due to its communication dependency on the security parameter λ. However, as n increases, this discrepancy diminishes. Indeed, when $n = 256$, ROTL's communication closely matches FLUTE and FLUTE+.

Concerning total runtime, ROTL and FLUTE+ align closely in both LAN (Figure 14(b)) and WAN settings (Figure 14(c)). FLUTE significantly lags behind others, primarily due to its $O(l \cdot n^2)$ local computation. When $n = 256$, it is approximately $11.6\times$ and $7.2\times$ slower than ROTL in LAN and WAN respectively. SP-LUT exhibits slightly faster performance in LAN, as its communication weakness is less pronounced in this setting. Conversely, in a WAN setting, it is $2.7\times$ slower than ROTL when $n = 256$.

The online communication and runtime (Figure 14(d)-14(f)) of these protocols demonstrate similar trends as observed in the total communication and runtime. Figure 14(g)-14(i) show the preprocessing communication and runtime. SP-LUT and FLUTE achieve better performance in the preprocessing phase as their primary overhead is in the online phase.

Figure 15 shows the evaluation results with the same table length and various output-bit lengths. The total and online overheads are roughly in similar trends with those in Figure 14. The preprocessing overheads for these protocols are stable, as the preprocessing phases of these protocols are independent of l.

In summary, ROTL and FLUTE+ achieve a commendable equilibrium between communication and computation in both online and total performance aspects. When $n = 256$ and $l = 64$,



(a) Total Communication. (b) Total time (LAN). (c) Total time (WAN).

(d) Online communication. (e) Online time (LAN). (f) Online time (WAN).

(g) Preprocessing communication. (h) Preprocessing time (LAN). (i) Preprocessing time (WAN).

Fig. 14: LUT evaluation with various table lengths n.

- ROTL involves 0.23KB total communication (0.03KB during online), 41.8μs runtime in LAN (3.1μs during online), and 68.9μs runtime in WAN (8.7μs during online), achieving upto 155× speedup in online runtime and 11.6× speedup in total runtime over FLUTE;
- FLUTE+ involves 0.17KB total communication (0.002KB during online), 44.6μs runtime in LAN (0.5μs during online), and 61.7μs runtime in WAN (1.5μs during online), achieving upto 962× speedup in online runtime and 10.8× speedup in total runtime over FLUTE.

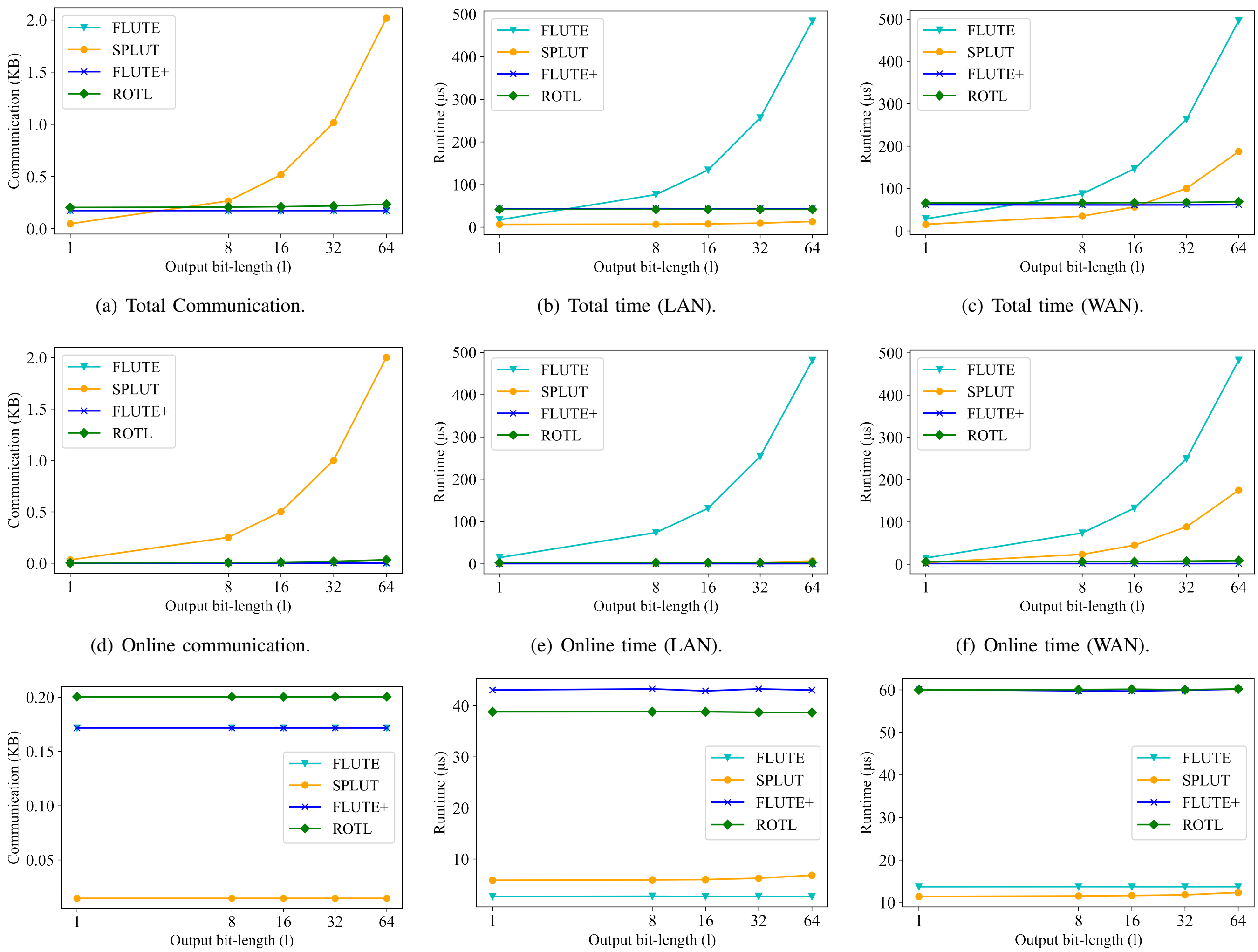
D. Evaluation of the secure comparison protocols

E. Secure LLM inference

We implemented the protocols for securely computing Softmax and GELU following the specifications in [16]. We first employ SP-LUT as the underlying LUT protocol and measure the performance as a baseline. Notice that SP-LUT is the most prevalent LUT used in secure inference (FLUTE cannot support arithmetic shares). Then, we replace SP-LUT with ROTL and assess the improvements.

Following Cheetah [17] and CrypTFlow2 [30], we left-shift the floating point numbers for $L = 12$ bits and drop the fractional part. During the inference, we use secure truncation to make sure the largest value is smaller than $2^l - 1$ with $l = 37$.

Table VI presents the comparison results, showcasing significant improvements over the baseline. There is a 4.8× reduction in total communication for Softmax and a 4.1× reduction for GELU. The improvements become even more pronounced when considering online communication, with a 1.5× reduction for Softmax and a 1.9× reduction for GELU. The improvements in LAN time are neutral, given the fast communication in LAN environments and the lightweight computation of SP-LUT. However, the improvements become pronounced again when considering WAN time, resulting in



(a) Total Communication. (b) Total time (LAN). (c) Total time (WAN).

(d) Online communication. (e) Online time (LAN). (f) Online time (WAN).

(g) Preprocessing communication. (h) Preprocessing time (LAN). (i) Preprocessing time (WAN).

Fig. 15: LUT evaluation with various output-bit lengths.

TABLE V: CMP

CMP ($\mathbb{Z}_{2^{37}}$)	Communication (MB)			LAN time (s)			WAN time (s)		
	total	online	prepr.	total	online	prepr.	total	online	prepr.
[17]	8.12	4.72	3.40	1.79	0.08	1.71	4.86	2.25	2.61
ROTL based CMP	75.61	1.93	73.68	3.78	0.07	3.71	13.12	2.11	11.01

savings of 35 seconds for Softmax and 1.7 minutes for GELU.

VIII. Related Work

In this section, we provide a succinct overview of related work.

Garbled LUTs. Yao's garbled circuits (GC) [32] is a generic protocol for secure two-party computation. In Yao's GC setting, earlier studies observed that 2-input/1-output gates can be extended into multi-input/multi-output gates, thereby reducing the overhead associated with circuit evaluation [22], [15], [24]. This could be considered as a general solution for LUT evaluations. Fairplay [22] supports garbled gates with up to 3-inputs, with their approach generalizing to an arbitrary number of inputs. TASTY [15] supports multi-input garbled gates, incorporating garbled-row reduction. More recently, [24] introduced garbled circuits featuring multi-input/multi-output gates.

LUTs w/wo preprocessing. The preprocessing model is widely used in secure multiparty computation (MPC) [2], [8]. It splits the computation into an input-independent preprocessing phase and an input-dependent online phase. In more detail, it enables the parties to generate correlated randomness in the preprocessing phase, subsequently expediting the online

TABLE VI: Softmax and GELU

Softmax $\left((\mathbb{Z}_{2^{37}}^{256\times256} \leftarrow \mathbb{Z}_{2^{37}}^{256\times256}) \times 12\right)$	Communication (MB)			LAN time (s)			WAN time (s)		
	total	online	prepr.	total	online	prepr.	total	online	prepr.
SP-LUT solution	1191.78	1133.62	58.16	24.78	8.59	16.187	139.53	115.05	24.48
ROTL based solution	247.2	189.56	57.64	19.87	3.42	16.45	53.61	29.79	23.82

GELU $\left(\mathbb{Z}_{2^{37}}^{256\times3072} \leftarrow \mathbb{Z}_{2^{37}}^{256\times3072}\right)$	Communication (MB)			LAN time (s)			WAN time (s)		
	total	online	prepr.	total	online	prepr.	total	online	prepr.
SP-LUT based solution	1837.05	1775.72	61.33	31.24	10.37	20.87	199.44	169.93	29.51
ROTL based solution	446.89	264.66	182.23	42.04	4.08	37.96	88.99	33.50	55.49

phase in terms of communication, interactive rounds, as well as overall runtime.

Ishai et al. [19] proposed a LUT protocol named OTTT based on the preprocessing model. It generates secret-shares of a rotated table in the preprocessing phase, by evaluating a Boolean circuit prepresenting the table once for every possible input. Dessouky et al. [9] proposed OP-LUT, which further reduces the cost of OTTT's preprocessing phase by leveraging OT instances. However, it still requires expensive circuit evaluations. In contrast, our proposed protocol for secret-shared rotations (cf. Figure 10) is significantly more lightweight.

In the same paper, Dessouky et al. [9] proposed another LUT protocol named SP-LUT, which operates without relying on the preprocessing model. Indeed, it only prepares $\log n$ random OTs during the preprocessing phase. SP-LUT is arguably the most lightweight protocol in terms of computation, but it incurs the highest communication cost as it necessitates transferring the entire table, unlike other LUT protocols that operate on a bit vector. Compounding this, such expensive communication occurs during the online phase.

Before our work, FLUTE achieved the optimal balance between overall and online performance. Indeed, the authors of FLUTE claimed that "*FLUTE matches or even outperforms the online performance of all prior approaches, while being competitive in terms of overall performance with the best prior LUT protocols* [4]". In this paper, we take a substantial leap forward, achieving a remarkable 962× speedup in online performance and a 10.8× speedup in overall performance. Additionally, we overcome FLUTE's limitation by enabling support for arithmetic shares.

LUT for secure inference. LUT is an important building block for computing non-linear functions in secure inference. CrypTFlow2 [30] employs LUT to realize the state-of-the-art Π_{CMP}, which is subsequently utilized to implement the ReLU activation function. SecFloat [27] leverages LUT for floating-point computation, striking a commendable balance between efficiency and accuracy. SIRNN [29] pioneers the exploration of employing LUT for computing complex functions such as sigmoid and softmax. Iron [14] applies them to LLM inference, but it exhibits high costs in communication and computation. CipherGPT [16] introduces an innovative method for computing GELU and softmax, optimizing the utilization LUT. Nevertheless, LUT evaluation remains a bottleneck. All these works use SP-LUT as the underlying LUT protocol, and ROTL could serve as a better alternative. SIGMA [13] uses function secret-sharing (FSS) to achieve a similar goal with LUT, but this solution relies on a trusted dealer.

IX. Conclusion

In response to the privacy concerns raised by LLM, we propose ROTL, a LUT protocol that is designed for secure LLM inference. It achieves upto 11.6× speedup in terms of overall performance and 155× speedup in terms of online performance over the state-of-the-art. Central to ROTL is a novel protocol for secret-shared rotation that is of independent interest. Furthermore, we propose another LUT protocol named FLUTE+ for boolean shares. It achieves upto 10.8× speedup in terms of overall performance and 962× speedup in terms of online performance over the state-of-the-art.

References

[1] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 535–548, New York, NY, USA, 2013. Association for Computing Machinery.

[2] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 1991.

[3] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 291–308. ACM, 2019.

[4] 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, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[5] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. Secret-shared shuffle. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020*, pages 342–372, Cham, 2020. Springer International Publishing.

[6] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[7] Geoffroy Couteau, Peter Rindal, and Srinivasan Raghuraman. Silver: Silent vole and oblivious transfer from hardness of decoding structured ldpc codes. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021*, pages 502–534, Cham, 2021. Springer International Publishing.

[8] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[9] Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni, and Michael Zohner. Pushing the communication barrier in secure computation using lookup tables. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[10] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 201–210. JMLR.org, 2016.

[11] Chun Guo, Jonathan Katz, Xiao Wang, and Yu Yu. Efficient and secure multiparty computation from fixed-key block ciphers. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 825–841, 2020.

[12] Xiaojie Guo, Kang Yang, Xiao Wang, Wenhao Zhang, Xiang Xie, Jiang Zhang, and Zheli Liu. Half-tree: Halving the cost of tree expansion in cot and dpf. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages 330–362, Cham, 2023. Springer Nature Switzerland.

[13] 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*, 2023.

[14] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. Iron: Private inference on transformers. In *NeurIPS*, 2022.

[15] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 451–462. ACM, 2010.

[16] Xiaoyang Hou, Jian Liu, Jingyu Li, Yuhan Li, Wen-jie Lu, Cheng Hong, and Kui Ren. Ciphergpt: Secure two-party gpt inference. *Cryptology ePrint Archive*, 2023.

[17] 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, Boston, MA, August 2022. USENIX Association.

[18] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2003.

[19] Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. On the power of correlated randomness in secure computation. In Amit Sahai, editor, *Theory of Cryptography*, pages 600–620, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[20] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669, Baltimore, MD, August 2018. USENIX Association.

[21] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 619–631. ACM, 2017.

[22] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In Matt Blaze, editor, *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 287–302. USENIX, 2004.

[23] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2505–2522. USENIX Association, 2020.

[24] Erik Pohle, Aysajan Abidin Aishajiang, and Bart Preneel. Poster: Fast evaluation of s-boxes in mpc. In *Network and Distributed System Security Symposium (NDSS 2022), Date: 2022/04/24-2022/04/28, Location: San Diego & online*, 2022.

[25] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[26] Srinivasan Raghuraman, Peter Rindal, and Titouan Tanguy. Expand-convolute codes for pseudorandom correlation generators from lpn. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023*, pages 602–632, Cham, 2023. Springer Nature Switzerland.

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

[28] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. Sirnn: A math library for secure RNN inference. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1003–1020. IEEE, 2021.

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

[30] 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*, CCS '20, page 325–342, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated ot with small communication. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1607–1626, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167. IEEE Computer Society, 1986.