ROTL: **Faster Lookup Table Evaluation**

Xiaoyang Hou*, Jian Liu*† ✉, Jingyu Li*, Jiawen Zhang*, Kui Ren*†, Chun Chen*†

* The State Key Laboratory of Blockchain and Data Security, Zhejiang University
† Hangzhou High-Tech Zone (Binjiang) Blockchain and Data Security Research Institute
{xiaoyanghou, liujian2411, jingyuli, kevinzh, kuiren, chenc}@zju.edu.cn

Abstract

Lookup table (LUT) is an important cryptography primitive, widely used in secure applications such as private set intersection, boolean circuit evaluation, and privacy-preserving machine learning. However, existing LUT constructions suffer from either high overhead or limited functionality.

In this paper, we propose ROTL, a secure two-party protocol for arithmetic LUT evaluation. Compared with SP-LUT (the state-of-the-art arithmetic LUT presented at NDSS '17), it achieves up to 3.3× speedup and 10.5× communication reduction in overall (preprocessing + online) and 21× speedup and 60× communication reduction in terms of the online phase.

At the heart of ROTL is a novel protocol for secret-sharing rotation, which allows two parties to generate additive secret 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 (USENIX '22), it achieves a 5× runtime speedup and 2.5× communication reduction in the online performance.

To support boolean secret sharing, we further provide an optimization (named FLUTE+) for FLUTE (the state-of-the-art boolean LUT presented at Oakland '23). For a boolean LUT with table size n and elements bit-width l, we reduce FLUTE's computation complexity from $O(n^2 l)$ to $O(n \log n + nl)$ and shift $O(n \log n)$ computation to the preprocessing phase without introducing communication overhead. As a result, FLUTE+ achieves up to 5× speedup in terms of overall (preprocessing and online) and over 600× speedup in terms of the online phase compared with FLUTE. The communication cost of FLUTE+ is exactly the same as FLUTE's in both the preprocessing phase and the online phase.

Keywords

privacy-preserving protocols, two-party secure computation.

1 Introduction

Secure computation [62] allows two or more parties to evaluate a public function while ensuring the correctness of the evaluation results and the privacy of each party's input. *Secure inference* [10, 23, 31, 38, 42, 44, 48, 56, 57] considers a scenario involving two semi-honest parties (a server $\mathcal{S}$ and a client C), which has emerged as a prominent topic. However, many functions in secure inference, such as non-linear operation, boolean circuit evaluation, and floating-point evaluation, are prohibitively expensive to realize with conventional gates in 2PC; instead, lookup tables (LUTs) have to be utilized [36, 54–56], making LUT an essential primitive in secure inference.

A LUT protocol allows two parties, holding a secret-shared index i, to learn secret-shared 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. More specifically, $\mathcal{S}$ and C generate correlated randomness during the preprocessing phase and consume it in the online phase.

The most common way (SP-LUT [18]) for arithmetic LUT evaluation is based on 1-out-of-n OT. Specifically, $\mathcal{S}$ generates a LUT output for each of C's n possible input shares, and masks these outputs with a single random number, which is $\mathcal{S}$'s output share. Then, C uses 1-out-of-n OT to get its output share. SP-LUT can be used for both arithmetic shares and boolean shares lookup table evaluation. Although this protocol is computationally efficient, it has to transfer the whole masked table during the online phase.

The state-of-the-art boolean 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 [10]. However, FLUTE involves expensive computations in the online phase, due to the evaluation of the multi-fan-in AND gates. Moreover, FLUTE has limited functionality and can only support boolean shares for its input and output. To be used with arithmetic shares, FLUTE has to be augmented with both arithmetic-to-boolean (A2B) and boolean-to-arithmetic (B2A) conversions, and the A2B conversion is particularly expensive for both computation and communication. As the result, using FLUTE with A2B and B2A in arithmetic secret share scenario is more expensive than using SP-LUT.

Our contribution. In this paper, we propose ROTL, an arithmetic LUT protocol that avoids transferring the whole table during the online phase. The rough idea of ROTL is to have 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 C and $\mathcal{S}$. Then, in the online phase, 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 [40] and OP-LUT [18]. 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 a natural generalization of the GMW [25] protocol. In contrast, we propose a novel protocol for table rotation that is significantly more lightweight. Recognizing that rotation

✉ Jian Liu is the corresponding author.

Proceedings on Privacy Enhancing Technologies 2026(2), 499–516

https://doi.org/10.56553/popets-2026-0058

is a special kind of permutation, we leverage the secret-shared permutation protocol proposed by Chase et al. [11], which is already quite lightweight, demanding only $n \log n$ random-OTs to shuffle a table of size n. By harnessing the inherent characteristics of rotation, we achieve a significant reduction in the number of necessary random-OTs, cutting it down to a mere $\log n$. Furthermore, we come up with a way allowing 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 $O(nl)$.

Boolean secret-shared LUTs are commonly used in boolean circuit evaluation. For completeness, we propose another boolean secret-shared LUT protocol named FLUTE+. It can be considered as an optimization of FLUTE [10]. For a boolean LUT with table size n and elements bit-width l, we reduce the computational complexity of FLUTE from $O(n^2 l)$ to $O(n \log n + nl)$ and shift $O(n \log n)$ computation to the preprocessing phase.

In addition to LUTs, *secure comparison* is another critical primitive for secure inference, extensively employed in privacy-preserving machine learning [36, 38, 41, 50] and private set intersection [32, 37]. 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 [38], it achieves a 5 × speedup and 2.5 × communication reduction in terms of online performance.

We summarize our contributions as follows:

- A novel 2PC protocol for secret-shared rotation, in Section 4;
- A novel arithmetic LUT protocol (named ROTL), which is up to 21× faster than SP-LUT (the state-of-the-art arithmetic LUT presented at NDSS '17), in Section 5;
- A novel boolean LUT protocol (named FLUTE+), achieving up to 600× speedup over FLUTE (the state-of-the-art boolean LUT presented at Oakland '23), in Section 6;
- A novel secure comparison protocol, which achieves a 5 × speedup and 2.5 × communication reduction compared with the state-of-the-art [38] in terms of online performance, in Section 7;
- A full-fledged implementation[1] and comprehensive benchmark, in Section 8.

2 Background and preliminaries

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

2.1 Notations

We use $\langle x \rangle^l = (\langle x \rangle^l_{\mathcal{S}}, \langle x \rangle^l_C)$ to denote 2-out-of-2 arithmetic secret share over $\mathbb{Z}_{2^l}$. Namely, $x = \langle x \rangle^l_{\mathcal{S}} + \langle x \rangle^l_C \mod 2^l$. Similarly, we use $\|x\|^l = (\|x\|^l_{\mathcal{S}}, \|x\|^l_C)$ to denote 2-out-of-2 boolean secret share, where $\oplus$ represents bit-wise xor and $x = \|x\|^l_{\mathcal{S}} \oplus \|x\|^l_C$. For simplicity, we omit the l notation of $\langle x \rangle^l$ and $\|x\|^l$ 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 $\mathbf{v}$. We use $\ggg$ to denote the cyclic right-rotate operation on vectors. We use 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 view^C_Π/$\mathsf{view}^{\mathcal{S}}_\Pi$ to denote the view of $C/\mathcal{S}$ when they run the protocol Π.

We set the security parameter λ to be 128 in this paper and denote the computation cost of correlation robust hash [27] as c_λ. Table 1 provides a summary of the frequently used notations in this paper.

Table 1: A table of frequent notations.

Notation	Description
C	client
$\mathcal{S}$	server
$\approx_c$	computational indistinguishability
λ	security parameter
c_λ	computation cost of correlation robust hash
n	table length
l	bit-length of each element in the table
$\ggg$	cyclic right-rotate operation on vectors
s	rotation offset
$\langle x \rangle^l$	$(\langle x \rangle^l_{\mathcal{S}}, \langle x \rangle^l_C)$ s.t. $x = \langle x \rangle^l_{\mathcal{S}} + \langle x \rangle^l_C \mod 2^l$
$\|x\|^l$	$(\|x\|_{\mathcal{S}}, \|x\|_C)$ s.t. $x = \|x\|_{\mathcal{S}} \oplus \|x\|_C$
$\|\mathsf{rOT}\|$	ammortized communication cost of 1-out-of-2 random OT
$\|\mathsf{MT}\|$	ammortized communication cost of multiplication triplet
$\mathcal{F}_{\mathsf{LUT}}$	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 $b = 0$

2.2 Lookup table (LUT)

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 $\langle id \rangle \in \mathbb{Z}_n$ as inputs, and returns the secret-shared entry $\langle x_{id} \rangle$. Figure 1 describes this functionality. We abuse the secret share notation here for simplicity, the boolean LUT protocol will be presented in Section 6.

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

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

Input: Secret-shared index

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

Output: Secret-shared entry

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

Figure 1: Ideal functionality for LUT.

[1] Our code is open source, available at https://github.com/zju-abclab/faster-LUT

2.3 Oblivious transfer (OT)

Oblivious Transfer (OT) [2, 6, 7, 19, 39, 51, 53, 61] allows a sender to send a message of the receiver's choice to the receiver while ensuring the sender learns nothing about the choice and the receiver learning nothing about the other messages.

In this paper, we adapt Silent OT [6, 7, 53, 61] which is based on the LPN (Learning Parity Error) assumption [4]. Silent OT produces a large number of 1-out-of-2 random OTs in batch in preprocessing, where the messages and choices are randomly sampled by the protocol. These random OTs can then be consumed to implement standard OT and other protocols [3] in the online phase. We denote the amortized communication cost per random OT as $|\mathsf{rOT}| \approx 0.6$ bits [61].

2.4 Boolean multiplication triplet

A boolean multiplication triplet is a tuple of three secret shared bits $\mathsf{MT} = (\|a\|, \|b\|, \|c\|)$ such that $c = a \wedge b$. A boolean multiplication triplet can be implemented by 2 parallel executions of 1-out-of-2 random OTs [2] and can be consumed to evaluate a AND gate [3]. We denote the amortized communication cost per boolean multiplication triplet as $|\mathsf{MT}| = 2|\mathsf{rOT}| \approx 1.2$ bits and the amortized communication cost per AND gate as $(4 + |\mathsf{MT}|) \approx 5.2$ bits.

2.5 Pseudo-random generator (PRG)

A Pseudo-Random Generator (PRG) is a deterministic function $G : \{0,1\}^n \to \{0,1\}^m$ which stretches a short random seed into a longer output which is polynomial-time-indistinguishable from a uniform distribution.

In this paper, PRG is instantiated using a correlation robust hash function from [27]. We denote the computation cost of a correlation robust hash [27] as c_λ.

2.6 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 [6, 11, 58], which, on input $i \in \mathbb{Z}_n$ from $\mathcal{S}$, allows $\mathcal{S}$ and C to jointly generate a vector $\mathbf{v}$ with randomly sampled elements, s.t., $\mathcal{S}$ obtains all elements in $\mathbf{v}$ except for v_i (denoted by $\mathbf{v}^{\mathcal{S}}$), and C obtains the whole vector $\mathbf{v}$ (denoted by $\mathbf{v}^{C}$) without learning the index i. Specifically, it has the following properties:

- *Correctness:* $v_j^C = v_j^{\mathcal{S}} \; \forall \; j \neq i$.
- *Position hiding:* a compromised 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 $\mathbf{v}^C$, cannot differentiate between the following two executions:
  - where $\mathbf{v}^C$ is generated according to PPRF;
  - where $\mathbf{v}^C$ is generated according to PPRF, but v_i^C is replaced with a random element from the domain.

In this paper, we adapt PPRF-GGM construction, which is proposed in [6, 11, 58] and optimized in [28]. In this construction, one PPRF with output vector length n requires $\log n$ parallel executions of 1-out-of-2 random OTs. The communication cost is $|\mathsf{rOT}| \cdot \log n$ bits and the computation cost is $n \cdot c_\lambda$ (c_λ represents correlation robust hash). For detailed protocol, we refer to the original paper, for it is slightly complicated and beyond our contribution. The security (position and value hiding property) of PPRF is based on Oblivious Transfer and PRG. We abuse the notion and use PPRF to denote the PPRF-GGM based $(n-1)$-out-of-n OT in this paper for simplicity.

2.7 Secret-shared permutation

Chase et al. [11] propose a protocol for secret-shared permutation, which allows two parties to learn secret shares of a permuted array $\mathbf{x} \in \mathbb{Z}_{2^l}^n$. Figure 2 shows the ideal functionality for this protocol: it takes an array $\mathbf{x}$ from C and a permutation π from $\mathcal{S}$, and returns secret shares $\langle \pi(\mathbf{x}) \rangle$ of a permuted array. We denote the permuted vector by $\pi(x) = (x_{\pi(0)}, ..., x_{\pi(n-1)})$.

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

Input:

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

Output:

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

Figure 2: Ideal functionality for secret-shared permutation.

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

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

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

The security is based on PPRF and PRG. The total communication cost of this protocol is $((|\mathsf{rOT}| + \lambda)n \log n + nl)$ bits and requires n^2 symmetric-key operations.

3 Threat model

We consider two non-colluding parties, $\mathcal{S}$ and C, and our proposed protocol is secure against a single semi-honest (passive) non-adaptive corruption [15, 24]. In this scenario, the probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ attempts to learn extra

Protocol Π_{Permut}

(1) 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 C and $\mathcal{S}$ respectively ($\mathcal{S}$ fills the punctured positions in $\mathbf{v}_i$ with 0s).

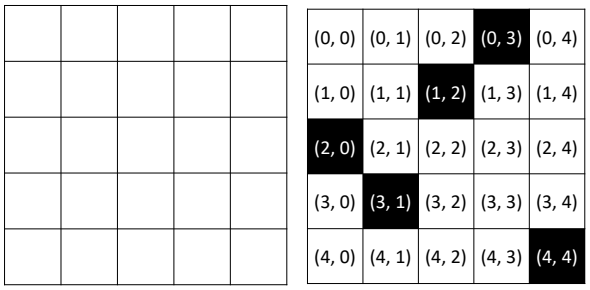
(2) For $i \in \mathbb{Z}_n$, C sets $r_i := \sum_j u_{i,j}$, $a_i := \sum_j u_{j,i}$.

(3) For $i \in \mathbb{Z}_n$, $\mathcal{S}$ sets $c_i := (\sum_j v_{j,\pi(i)}) - (\sum_j v_{i,j})$.

(4) C sends $\mathbf{x} - \mathbf{a}$ to $\mathcal{S}$ and outputs $\mathbf{r}$.

(5) $\mathcal{S}$ outputs $\pi(\mathbf{x} - \mathbf{a}) + \mathbf{c}$.

Figure 3: The secret-shared permutation protocol.



(a) C receives the full matrix.

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

Figure 4: Visualization of the two matrices received by C and $\mathcal{S}$ after Step 1 of Π_{Permut}, with $\pi = (3, 2, 0, 1, 4)$ and the dark blocks denote puncture positions.

information from legitimate messages but must strictly follow the protocol's procedure. We assume the adversary is computationally bounded and we set the computational security parameter λ to 128.

The security proof is based on the Universal Composability (UC) framework [43]. Our proposed protocols are based on Oblivious Transfer and Pseudo-Random Generator that have already been proven secure.

4 Secret-shared rotation

In this section, we provide a protocol for secret-shared rotation, the ideal functionality of which is shown in Figure 5.

As rotation is a special kind of permutation, we can directly employ the secret-shared permutation protocol described in Section 2.7, with $\mathcal{S}$ using

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

as the input permutation.

In Figure 6, a matrix is depicted as a replacement for Figure 4(b) when permutation is substituted with rotation. Clearly, this matrix is more regular. In the i-th row, the $\pi(i)$-th position is punctured

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

Input:

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

Output:

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

Figure 5: Ideal functionality for secret-shared rotation.

(0, 0)	(0, 1)	(0, 2)	(0, 3)	(0, 4)
(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 4)
(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)
(3, 0)	(3, 1)	(3, 2)	(3, 3)	(3, 4)
(4, 0)	(4, 1)	(4, 2)	(4, 3)	(4, 4)

Figure 6: $\mathcal{S}$ receives "punctured" matrix that replaces Figure 4(b) when rotation is applied with $s = 2$.

with $\pi(i) = (i + s) \mod n$. We observe that in each row (except the first), the punctured position is rotated one step right from the previous one. So all the punctured positions can be interpreted as a right slant column.

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 6 into a rhombus shape that is shown in Figure 7. More specifically, we transform the punctured matrix into n column vectors $(\mathbf{t}_0, \mathbf{t}_1, ...\mathbf{t}_{n-1})$, where $\mathbf{t}_{i,j} = \mathbf{v}_{j,(i+j) \bmod n}$.

For each slant column of the rhombus, we have C and $\mathcal{S}$ use the result of PPRF as PRG seed to generate elements in that column. We notice that the transform operation is independent of s, all the punctured positions will be rotated to the same slant column. Then, all the punctured elements can be generated by the same seed. To this end, we could have $\mathcal{S}$ use a single PPRF to get all PRG seeds except for the one that generates the punctured elements. The number of required PPRF instances is reduced from n to 1, and the communication cost is $((|\mathsf{rOT}| + \lambda) \cdot \log n)$ bits.

C and $\mathcal{S}$ could locally recover the matrix in Figure 6 using PRG, and then calculate the row-wise and column-wise sums of this matrix. However, it suffers from a $O(n^2)$ space complexity. Given that $\mathcal{F}_{\mathsf{Rotate}}$ is usually batching invoked, recovering such $n \times n$ matrices brings heavy memory overhead.

We provide an optimization to calculate the row-wise and column-wise sums without recovering the matrix. Denote $\mathbf{d}_i$ as a column of elements generated by C from expanding the i-th seed. The

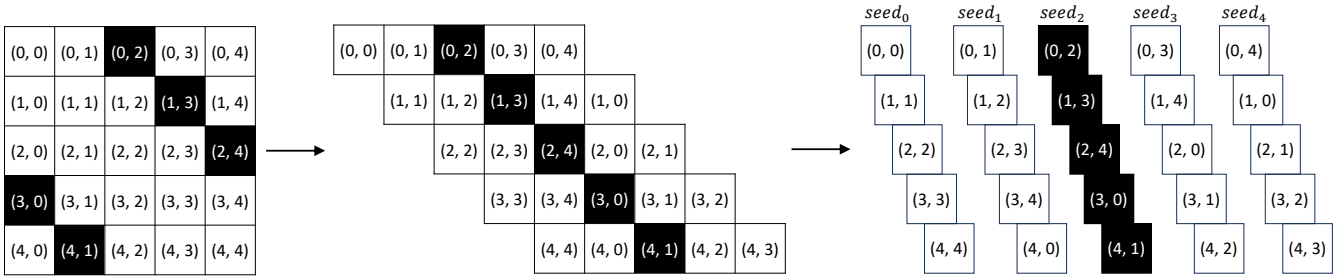


Figure 7: Transformation of Figure 6 such that the punctured elements can be generated by a single seed.

row-wise sums can simply be calculated as $\mathbf{r} := \sum_i \mathbf{d}_i$, as our transformation in Figure 7 does not change the sum of each row. To compute the column-wise sums, C right-rotates each $\mathbf{d}_i$ by i elements, that is shown in Figure 8. The row-wise sums of slant columns in this new rhombus (i.e., $\mathbf{a} := \sum_i(\mathbf{d}_i \ggg i)$) are exactly the column-wise sums of the matrix in Figure 6.

The detailed protocol is shown in Figure 9.

Theorem 1. *The protocol in Figure 9 securely realizes the ideal functionality $\mathcal{F}_{\text{Rotate}}$ in the $(\mathcal{F}_{\text{PRG}}, \mathcal{F}_{\text{PPRF}})$-hybrid model.*

The proof of Theorem 1 can be found in the Appendix A.

5 Lookup table with arithmetic secret share

In this section, we present our arithmetic secret share LUT protocol ROTL.

5.1 Strawman solution

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

$$\mathbf{y} := (\mathbf{x} \ggg \langle s\rangle_C).$$

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

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

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

In the online phase, we have C open $(\langle id\rangle_C + \langle s\rangle_C)$ and $\mathcal{S}$ open $(\langle id\rangle_{\mathcal{S}} + \langle s\rangle_{\mathcal{S}})$ to recover the value id'. C outputs $\langle y'_{id'}\rangle_C$ and $\mathcal{S}$ outputs $\langle y'_{id'}\rangle_{\mathcal{S}}$. We notice that s and $\mathbf{y}'$ can only be used as one-time pad for security, which means it needs independent s and $\mathcal{F}_{\text{Rotate}}$ invocation for each ROTL.

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

5.2 ROTL

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

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

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

In the online phase, C and $\mathcal{S}$ recover $s' := (id - s) \mod n$. Then, they locally right-rotate $\langle \mathbf{b}'\rangle_C$ and $\langle \mathbf{b}'\rangle_{\mathcal{S}}$ by s' elements, resulting in $\langle \mathbf{b}''\rangle_C$ and $\langle \mathbf{b}''\rangle_{\mathcal{S}}$. The id-th bit in $\mathbf{b}''$ is 1 and other bits are 0s. The element to be chosen is equal to the dot-product between $\mathbf{b}''$ and $\mathbf{x}$, the secret shares of which can be computed locally by 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., $\mathbb{Z}^n_{2^l}$) 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, 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 $(\langle b''_i\rangle^1_C, \langle b''_i\rangle^1_{\mathcal{S}})$ can be either $(0,0)$ or $(1,1)$, meaning that:
$$\langle b''_i\rangle^1_C = \langle b''_i\rangle^1_{\mathcal{S}} \text{ when } b'_i = 0.$$
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 $(\langle b''_i\rangle^l_C, -\langle b''_i\rangle^l_{\mathcal{S}})$:
$$\langle b''_i\rangle^l_C + (-\langle b''_i\rangle^l_{\mathcal{S}}) \mod 2^l = 0.$$
- If $b''_i = 1$, the shares $(\langle b''_i\rangle^1_C, \langle b''_i\rangle^1_{\mathcal{S}})$ can be either $(1,0)$ or $(0,1)$. The above procedure for case "$b''_i = 0$" may result in an error:
  - if $(\langle b''_i\rangle^1_C, \langle b''_i\rangle^1_{\mathcal{S}}) = (1,0)$,
    $\langle b''_i\rangle^l_C + (-\langle b''_i\rangle^l_{\mathcal{S}}) \mod 2^l = 1$,
    which leads to a correct final output;
  - if $(\langle b''_i\rangle^1_C, \langle b''_i\rangle^1_{\mathcal{S}}) = (0,1)$,
    $\langle b''_i\rangle^l_C + (-\langle b''_i\rangle^l_{\mathcal{S}}) \mod 2^l = -1$,
    which flips the sign of the final output.

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

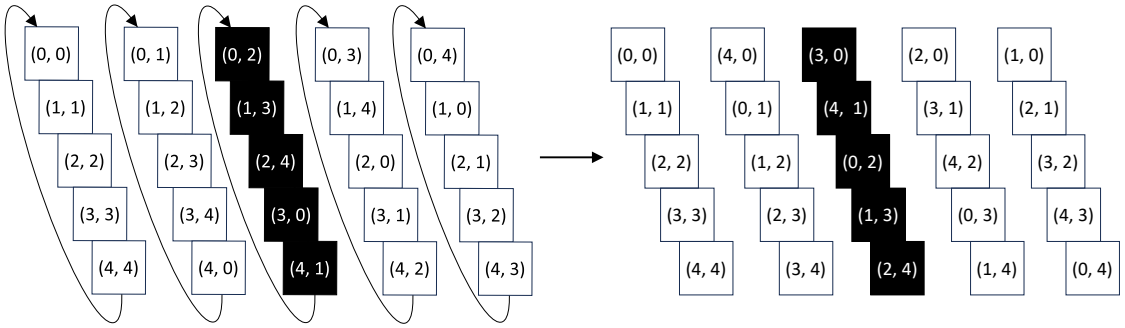


Figure 8: Transformation of Figure 7 such that the row sums of slant columns (right) are exactly the column sums of Figure 6.

Protocol Π_{Rotate}

(1) C and $\mathcal{S}$ run one execution of PPRF, where $\mathcal{S}$ uses s as its input. Let $\mathbf{seed}^C$ and $\mathbf{seed}^S$ be the output for C and $\mathcal{S}$ respectively.
(2) C runs as follows:
  (a) $\forall i \in \mathbb{Z}_n, \mathbf{d}_i \leftarrow \text{PRG}(seed_i^C)$, where $\mathbf{d}_i \in \mathbb{Z}_{2^l}^n$;
  (b) $\mathbf{r} := \sum_i \mathbf{d}_i$;
  (c) $\mathbf{a} := \sum_i (\mathbf{d}_i \ggg i)$;
(3) $\mathcal{S}$ runs as follows:
  (a) $\forall i \in \mathbb{Z}_n$ and $i \neq s, \mathbf{t}_i \leftarrow \text{PRG}(seed_i^S)$, where $\mathbf{t}_i \in \mathbb{Z}_{2^l}^n$;
  (b) $\mathbf{t}_s := [0 \ldots 0]$;
  (c) $\mathbf{r}^* := \sum_i \mathbf{t}_i$;
  (d) $\mathbf{a}^* := \sum_i (\mathbf{t}_i \ggg i)$;
  (e) $\mathbf{c} := (\mathbf{a}^* \ggg s) - r^*$.
(4) C sends $\mathbf{x} - \mathbf{a}$ to $\mathcal{S}$ and outputs $\mathbf{r}$.
(5) $\mathcal{S}$ outputs $((\mathbf{x} - \mathbf{a}) \ggg s) + \mathbf{c}$.

Figure 9: The secret-shared rotation protocol.

– if $(\langle b''_{id} \rangle_C^1, \langle b''_{id} \rangle_{\mathcal{S}}^1) = (0, 1)$, $sum = -1$.

Then, we only need to multiply sum to the dot-product.

The detailed protocol is shown in Figure 10.

Theorem 2. *The protocol in Figure 10 securely realizes the ideal functionality $\mathcal{F}_{\text{LUT}}$ in the $(\mathcal{F}_{\text{Rotate}}, \mathcal{F}_{\text{Mult}})$-hybrid model.*

The proof of Theorem 2 can be found in the Appendix B.

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 interpreted as $\langle s \rangle_{\mathcal{S}}$. As a result, we reduce the communication to $((|\text{rOT}| + \lambda) \log n + n)$ bits.

Protocol Π_{ROTL}

Parameter: Public table $\mathbf{x} \in \mathbb{Z}_{2^l}^n$
Input: Secret-shared index
- C: $\langle id \rangle_C \in \mathbb{Z}_n$
- $\mathcal{S}$: $\langle id \rangle_{\mathcal{S}} \in \mathbb{Z}_n$

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

Online:
(1) C sends $(\langle id \rangle_C - \langle s \rangle_C)$ to $\mathcal{S}$,
$\mathcal{S}$ sends $(\langle id \rangle_{\mathcal{S}} - \langle s \rangle_{\mathcal{S}})$ to C, simultaneously.
C and $\mathcal{S}$ recover $s' := (id - s)$.
(2) C locally right-rotates $\langle \mathbf{b}' \rangle_C$ by s' elements, resulting in $\langle \mathbf{b}'' \rangle_C$, and computes $\langle z \rangle_C^l := \sum_i (\langle b''_i \rangle_C^l \cdot x_i)$.
(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 $\langle z \rangle_{\mathcal{S}}^l := \sum_i (-\langle b''_i \rangle_{\mathcal{S}}^l \cdot x_i)$.
(4) C and $\mathcal{S}$ run $(\langle z' \rangle_C^l, \langle z' \rangle_{\mathcal{S}}^l) \leftarrow \mathcal{F}_{\text{Mult}}(sum, z)$.

Output:
(1) C outputs $\langle z' \rangle_C^l$.
(2) $\mathcal{S}$ outputs $\langle z' \rangle_{\mathcal{S}}^l$.

Figure 10: The ROTL protocol.

Another optimization is to replace $\mathcal{F}_{\text{Mult}}$ with $\mathcal{F}_{\text{MUX}}$ at Step 4 (in the online phase), as $\mathcal{F}_{\text{MUX}}$ can be realized with two executions of

1-out-of-2 random OT [57], much cheaper than $\mathcal{F}_{\mathsf{Mult}}$. The fundamental idea of this optimization is:

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

where β is the sign bit of *sum*.

Given that $sum = 1$ or -1, we could even locally get β for free (without invoking $\mathcal{F}_{\mathsf{CMP}}$). This is based on the inherent property of the arithmetic share architecture. Let α_i^C be the i-th bit of $\langle sum \rangle_C^l$ and α_i^S be the i-th bit of $\langle sum \rangle_S^l$, with $i = 0$ denoting the least-significant bit. Let c_i be the carry-bit from $(i-1)$-th bit where $i \geq 1$, $c_i = 1$ iff. $\alpha_i^C = 1$ and $\alpha_i^S = 1$. Then, the i-th bit of *sum* is equivalent to:

$$\alpha_i^C \oplus \alpha_i^S \oplus c_{i-1}$$

where $i \geq 1$. The first observation is $\mathsf{LSB}(sum) = 1$ and $c_0 = 0$ for both $sum = 1$ and $sum = -1$. Now we concentrate on the second least significant bit of *sum*. In the case of $sum = 1 = (0 \ldots 01)_2$, we have $\left(\alpha_1^C + \alpha_1^S + c_0\right) \mod 2 = 0$, which means $\alpha_1^C \oplus \alpha_1^S = 0$. In the case of $sum = -1 = (1 \ldots 11)_2$, we have $\left(\alpha_1^C + \alpha_1^S + c_0\right) \mod 2 = 1$, which means $\alpha_1^C \oplus \alpha_1^S = 1$. Therefore, we can directly use α_1^C and α_1^S as the input shares of β for $\mathcal{F}_{\mathsf{MUX}}$.

5.3 Comparison

Table 2 provides a theoretical comparison between ROTL and existing LUT protocols that support arithmetic shares.

We use $|\mathsf{rOT}|$ and $|\mathsf{MT}|$ to denote the communication costs for generating a 1-out-of-2 random OT and a boolean multiplication triplet, respectively. A boolean multiplication triplet can be generated from two 1-out-of-2 random OTs using Beaver's multiplication method [3], which yields $|\mathsf{MT}| = 2|\mathsf{rOT}|$. Notice that we assume a different $|\mathsf{MT}|$ with those in OP-LUT [18] and FLUTE[10]:

- OP-LUT [18] assumed a relatively large $|\mathsf{MT}|$ (i.e., 138 bits), as they use IKNP [2, 39] for oblivious transfer extensions.
- FLUTE [10] replaced IKNP with silent OT extension [6], which reduces $|\mathsf{rOT}|$ to around 0.118 bits, but requires more computation.
- We use Ferret-OT [61], which achieves a better trade-off between communication and computation. The communication cost per random-OT is around 0.6 bits and $|\mathsf{MT}| \approx 1.2$ bits.

For alignment, we use $|\mathsf{rOT}|$ and $|\mathsf{MT}|$ in our theoretical analysis.

In terms of communication, OTTT [40] and OP-LUT [18] only need to transfer $2 \log n$ bits in online phase, but involve a communication complexity of $O(nl \log n)$ and $O(n^2 l)$ respectively during preprocessing. SP-LUT [18] reduces the total communication, but involves transferring the whole table in the online phase. The preprocessing and online communication complexities of SP-LUT are $O(\log n)$ and $O(nl)$, respectively. Our proposed ROTL has a total communication cost of $O(\lambda \log n + l)$, which is much cheaper than OTTT and OP-LUT. In fact, it is the first arithmetic secret-sharing LUT protocol with communication overhead independent of nl.

Table 3 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 $2n^2 l + (n^2 + n) \cdot c_\lambda$ during preprocessing and $nl + 2l$ during online computation:

- It requires $n \cdot c_\lambda$ for PPRF and $n^2 \cdot c_\lambda$ for generating the rhombus of Figure 7.
- It requires $2n^2 l$ to compute the row- and column-wise sums of the rhombus in Figure 9.
- During online computation, it requires a dot-product of the public table with a boolean vector, which is equivalent to nl bits computation, and only plaintext addition is required. Additionally, it requires one $\mathcal{F}_{\mathsf{MUX}}$ operation, which is equal to $2l$ bits computation.

We refer to Section 8 for a detailed empirical comparison.

6 Lookup table with boolean secret share

Arguably, ROTL achieves the best tradeoff between computation and communication, but it necessitates augmentation with A2B and B2A conversions to support boolean secret shares. For boolean circuit evaluation, we introduce another boolean secret share LUT protocol named FLUTE+. It can be considered as an optimization of FLUTE [10]. For a boolean secret share LUT with table size n and elements bit-width l, we reduce the computational complexity from $O(n^2 l)$ to $O(n \log n + nl)$, and shift 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 $\log n$ 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, C and $\mathcal{S}$ generate secret shares of $\log n$ random bits (α_0 and α_1 in this case), and use $\mathcal{F}_{\mathsf{AND}}$ to compute the secret shared AND sum of all subsets of these random bits ($\beta := \alpha_0 \wedge \alpha_1$ in this case). In the online phase, they reveal $m_0 := \alpha_0 \oplus x_0$ and $m_1 := \alpha_1 \oplus x_1$. Then, y_i can be expanded as several binomial expansions. We take the first component of y_i as an example, which is:

$$\begin{aligned}
&(\overline{x}_0 \wedge \overline{x}_1 \wedge a_i) \\
&= (\overline{m}_0 \oplus \alpha_0) \wedge (\overline{m}_1 \oplus \alpha_1) \wedge a_i \\
&= (\overline{m}_0 \wedge \overline{m}_1 \oplus \overline{m}_0 \wedge \alpha_1 \oplus \overline{m}_1 \wedge \alpha_0 \oplus \alpha_0 \wedge \alpha_1) \wedge a_i \\
&= \ (\overline{m}_0 \wedge \overline{m}_1) \wedge a_i \\
&\quad \oplus (\overline{m}_0 \wedge \|\alpha_1\|_C \oplus \overline{m}_1 \wedge \|\alpha_0\|_C \oplus \|\beta\|_C) \wedge a_i \\
&\quad \oplus (\overline{m}_0 \wedge \|\alpha_1\|_S \oplus \overline{m}_1 \wedge \|\alpha_0\|_S \oplus \|\beta\|_S) \wedge a_i,
\end{aligned}$$

the secret-share of which can be computed locally by $\mathcal{S}$ and C. For the public values such as $\overline{m}_0 \wedge \overline{m}_1$, C holds $\overline{m}_0 \wedge \overline{m}_1$ and $\mathcal{S}$ holds 0. Recall that each binomial expansion contains $n = 2^{\log n}$ terms, and each output bit contains n components, so the online computation for each output bit is $O(n^2)$. FLUTE [10] runs the above process

Table 2: Comparison with existing LUT protocols that support arithmetic inputs. A table has n l-bit elements. We use Ferret-OT [61] for random 1-out-of-2 Oblivious Transfer, which roughly requires $|\mathsf{rOT}| \approx 0.6$ bits communication cost per random OT. The multiplication triplet is implement by 2 random OT instances, so we have $|\mathsf{MT}| = 2|\mathsf{rOT}| \approx 1.2$ bits communication cost. When calculating the computational overhead, we only consider one party, as the two parties can run in parallel. We use c_λ to denote the computation cost of correlation robust hash [27].

Protocol	Communication (bits)		Computation	
	preprocessing	online	preprocessing	online
OTTT [40]	$(\|\mathsf{MT}\|+4)(\log n-1)nl$	$2\log n$	$ln\log n$	$\log n$
OP-LUT [18]	$\|\mathsf{rOT}\|\cdot\log n+n^2l$	$2\log n$	$n^2l\log n$	$\log n$
SP-LUT [18]	$\|\mathsf{rOT}\|\cdot\log n$	$\log n+nl$	$n\log n$	$\log n+nl+n\cdot c_\lambda$
GC-LUT [34]	$nl+l\lambda\log n$	-	$nl\lambda+(n+l\log n+\frac{nl}{\lambda})\cdot c_\lambda$	-
MAESTRO [49]	$2\sqrt{n}-\log n-2$	$2\log n$	$O(n\log n)$	$O(n)$
Scalable MPC [21]	$O(nl)$	$2\log n$	$O(nl)$	$\log n$
GROTTO [59]	$2\lambda\log n$	$2\log n$	$O(\log n)$	$nl+\log n\cdot c_\lambda$
ROTL (Ours)	$(\|\mathsf{rOT}\|+\lambda)\log n+n$	$2\log n+4l+2$	$2n^2l+(n^2+n)\cdot c_\lambda$	$\log n+(n+2)l$

Table 3: 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

for all l output bits, which involves $O(l\cdot n^2)$ online computation complexity.

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 constant across all output bits, hence we can compute them once and reuse the results. This optimization reduces the online computation of FLUTE from $O(n^2l)$ to $O(n^2+nl)$. We further observe that the online computation for each party (e.g., C) to compute an output bit is:

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

Although 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:

$$\begin{aligned}\|s_0\|_C&:=0\wedge0\oplus0\wedge\|\alpha_1\|_C\oplus0\wedge\|\alpha_0\|_C\oplus\|\beta\|_C,\\\|s_1\|_C&:=0\wedge1\oplus0\wedge\|\alpha_1\|_C\oplus1\wedge\|\alpha_0\|_C\oplus\|\beta\|_C,\\\|s_2\|_C&:=1\wedge0\oplus1\wedge\|\alpha_1\|_C\oplus0\wedge\|\alpha_0\|_C\oplus\|\beta\|_C,\\\|s_3\|_C&:=1\wedge1\oplus1\wedge\|\alpha_1\|_C\oplus1\wedge\|\alpha_0\|_C\oplus\|\beta\|_C.\end{aligned}$$

After knowing m_0 and m_1 in the online phase, it could locally rearrange 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 $O(nl)$ local computation online.

We further reduce the preprocessing computation from $O(n^2)$ to $O(n\log n)$ by leveraging the *butterfly diagram optimization* [16]. In more detail, we transform (s_0, s_1, s_2, s_3) to:

$$\begin{aligned}\|s_0\|_C&=(0\oplus\|\alpha_1\|_C)\wedge(0\oplus\|\alpha_0\|_C)\\&=\|\alpha_1\|_C\wedge\|\alpha_0\|_C,\\\|s_1\|_C&=(0\oplus\|\alpha_1\|_C)\wedge(1\oplus\|\alpha_0\|_C)\\&=\|\alpha_1\|_C\wedge(1\oplus\|\alpha_0\|_C),\\&=\|\alpha_1\|_C\oplus\|\alpha_1\|_C\wedge\|\alpha_0\|_C,\\\|s_2\|_C&=(1\oplus\|\alpha_1\|_C)\wedge(0\oplus\|\alpha_0\|_C),\\&=(1\oplus\|\alpha_1\|_C)\wedge\|\alpha_0\|_C,\\&=\|\alpha_0\|_C\oplus\|\alpha_1\|_C\wedge\|\alpha_0\|_C,\\\|s_3\|_C&=(1\oplus\|\alpha_1\|_C)\wedge(1\oplus\|\alpha_0\|_C),\\&=1\oplus\|\alpha_0\|_C\oplus\|\alpha_1\|_C\oplus\|\alpha_1\|_C\wedge\|\alpha_0\|_C.\end{aligned}$$

To compute $(\|s_0\|_C, \|s_1\|_C, \|s_2\|_C, \|s_3\|_C)$ efficiently, C first sets:

$$\begin{aligned}\|s_0\|_C&:=\|\alpha_0\|_C\wedge\|\alpha_1\|_C,\\\|s_1\|_C&:=\|\alpha_1\|_C,\\\|s_2\|_C&:=\|\alpha_0\|_C,\\\|s_3\|_C&:=1.\end{aligned}$$

Then, C computes:

$$\begin{aligned}\|s_3\|_C&:=\|s_3\|_C\oplus\|s_2\|_C,\\&=1\oplus\|\alpha_0\|_C.\\\|s_1\|_C&:=\|s_1\|_C\oplus\|s_0\|_C,\\&=\|\alpha_1\|_C\oplus\|\alpha_0\|_C\wedge\|\alpha_1\|_C\end{aligned}$$

Table 4: Comparison of Boolean LUT protocols in theoretical asymptotic complexity. The table size is n and the bit-width of each entry is l. We use Ferret-OT [61] for random 1-out-of-2 Oblivious Transfer, which roughly requires $|\mathsf{rOT}| \approx 0.6$ bits communication cost per random OT. The multiplication triplet is implement by 2 random OT instances, so we have $|\mathsf{MT}| = 2|\mathsf{rOT}| \approx 1.2$ bits communication cost. When calculating the computational overhead, we only consider one party, as the two parties can run in parallel.

Protocol	Communication (bits)		Computation (bits)	
	preprocessing	online	preprocessing	online
FLUTE [10]	$(\|\mathsf{MT}\|+4)(n-\log n-1)$	$2\log n$	$n-\log n-1$	$nl+n^2l$
FLUTE+	$(\|\mathsf{MT}\|+4)(n-\log n-1)$	$2\log n$	$n-\log n-1+n\log n$	$nl+n\log n$

Next, C computes:

$$\begin{aligned}\|s_2\|_C &:= \|s_2\|_C \oplus \|s_0\|_C,\\ &= \|\alpha_0\|_C \oplus \|\alpha_1\|_C \wedge \|\alpha_0\|_C,\\ \|s_3\|_C &:= \|s_3\|_C \oplus \|s_1\|_C,\\ &= 1 \oplus \|\alpha_0\|_C \oplus \|\alpha_1\|_C \oplus \|\alpha_0\|_C \wedge \|\alpha_1\|_C.\end{aligned}$$

In the general case, for $\log n$ input bits, the process for generating $(\|s_0\|_C, \|s_1\|_C, \|s_2\|_C, \cdots \|s_{n-1}\|_C)$ is as follows:

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

   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, $\|s_i\|_C := \|s_i\|_C \oplus \|s_{i+2^j}\|_C$.

$\mathcal{S}$ runs the same process, except that, in Step 1, $\mathcal{S}$ sets $\|s_i\|_{\mathcal{S}} := 0$ 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 4 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 overhead in re-arranging the order of $(s_0, \cdots, s_{n-1})$, amounting to $n \log n$ bits of computation.

7 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 C compute an arithmetic share of $a := x - y$. Then, b is the negation of most-significant bit (MSB) of a:

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

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

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

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

Let $c = \langle a'\rangle_C$ 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 11.

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

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

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

Figure 11: Ideal functionality for secure comparison.

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

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

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

With ROTL (cf. Section 5), all the expensive operations are shifted to the preprocessing phase. As the public table is binary, $\mathcal{F}_{\mathsf{MUX}}$ can be bypassed, leaving only a single dot-product operation in the online phase. However, this is feasible for relatively small $n = 2^l$, because ROTL requires $O(n)$ communication and $O(n^2)$ computation in the preprocessing phase. To this end, we use the same technique in [38, 57] to partition e into smaller blocks. Essentially, our protocol can be interpreted as replacing the SP-LUT in the protocols of Cheetah [38] with our proposed ROTL. Figure 12 shows the details of the millionaires' protocol.

8 Evaluation

In this section, we empirically compare ROTL and FLUTE+ with existing LUT protocols. We also compare our secure comparison

[2] We set the modular to 2^l, to make sure that $2^{l-1} < e < 2^l$ when $c < d$; and $0 \leq e < 2^{l-1}$ when $c \geq d$.
If the modular is 2^{l-1}, it has $0 < e < 2^{l-1}$ for both $c < d$ or $c > d$, which can not be distinguished with a public table.

Protocol Π_{MILL}

(1) 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 $m-1$, C and $\mathcal{S}$ compute $e_i = c_i - d_i \bmod 2^{l'+1}$, and invoke $\mathcal{F}_{\mathsf{LUT}}(\langle e_i \rangle_C, \langle e_i \rangle_{\mathcal{S}})$ with two public tables:
$$[\underbrace{0, ..., 0}_{2^{l'}}, \underbrace{1, ..., 1}_{2^{l'}}], \quad [\underbrace{1, 0, ..., 0}_{2^{l'+1}-1}]$$
and get the secret-shares of
$$\mathtt{lt}_i = \mathbf{1}\{c_i < d_i\} \text{ and } \mathtt{eq}_i = \mathbf{1}\{c_i = d_i\}$$
respectively. Notice that, since the inputs to both tables are identical (i.e., e_i), a single ROTL invocation is enough.
(3) C and $\mathcal{S}$ invoke $\mathcal{F}_{\mathsf{AND}}$ and output:
$$\mathtt{lt}_{m-1} \oplus (\mathtt{eq}_{m-1} \wedge \mathtt{lt}_{m-2}) \oplus$$
$$(\mathtt{eq}_{m-1} \wedge \mathtt{eq}_{m-2} \wedge \mathtt{lt}_{m-3}) \oplus ... \oplus$$
$$(\mathtt{eq}_{m-1} \wedge ... \wedge \mathtt{eq}_1 \wedge \mathtt{lt}_0)$$

Figure 12: The millionaires' protocol.

protocol with the state-of-the-art [38]. For privacy-preserving machine learning (PPML) application, we compare our ROTL based protocol with GROTTO [59].

8.1 Implementation

We fully implemented ROTL and FLUTE+ in C++ and set the security parameter $\lambda = 128$ for all implementations. We use [27] for PRG and use Ferret-OT [61] from EMP-toolkit[3] for random OT instances. For PPRF [6, 11, 58], we incorporated the half-tree [28] optimization to reduce both communication and computation cost. We use AVX2 (Advanced Vector Extensions) to accelerate the operations. We use AES-NI (Advanced Encryption Standard Instructions) to accelerate AES operation, which is used in circular correlation robustness hash [27] and PRG [27, 28]. For SP-LUT[4] [18], FLUTE[5] [10], Scalable-MPC[6] [21] and MAESTRO[7] [49] we use their open-source implementations.

8.2 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.

[3]EMP-toolkit: https://github.com/emp-toolkit/emp-ot
[4]SP-LUT: https://github.com/encryptogroup/ABY
[5]FLUTE: https://github.com/encryptogroup/FLUTE
[6]Scalable-MPC: https://github.com/radhika1601/ScalableMixedModeMPC
[7]MAESTRO: https://github.com/KULeuven-COSIC/maestro

To ensure a fair comparison, we present separate benchmarks for arithmetic LUT and boolean LUT. For arithmetic LUT, our comparison focuses on our proposed ROTL against SP-LUT, for SP-LUT has been demonstrated to outperform OTTT and OP-LUT [18] in terms of both communication and computation, as detailed in our theoretical comparison in Table 2. For boolean LUT, we compare our proposed FLUTE+ with the original FLUTE [10].

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.

8.3 Arithmetic LUT benchmark

The main bottleneck of ROTL is the preprocessing phase (recall that in Table 2). Specifically, ROTL needs to transfer $O(\lambda \log n)$ bits more than SP-LUT and has $O(n^2)$ correlation robust hash operation overhead in preprocessing phase. Table 5 shows that the preprocessing runtime of ROTL is 3.1× - 5.4× slower than SP-LUT. However, in the online phase, the communication and computation complexities of ROTL are both $O(\log n + l)$ and without any correlation robust hash operation, while SP-LUT needs $O(nl)$ communication and $O(n)$ correlation robust hash operation overhead. When the table size is relatively small, e.g., $n = 4$, ROTL's online communication and runtime are slightly higher than SP-LUT. As the table size increases, ROTL's online advantage becomes more pronounced, when $n = 256$ and $l = 64$, the biggest setting in our benchmark, ROTL is 2× and 20× faster than SP-LUT in LAN and WAN setting respectively, and reduces the online communication cost by over 60×.

Table 6 shows the improvement factor of online runtime comparison between ROTL and SP-LUT, we highlight the entries in blue color where ROTL outperforms SP-LUT. It can be seen that ROTL is faster than SP-LUT in the scenarios where the public table is relatively large or the network bandwidth is limited. ROTL's runtime improvement is mainly due to the reduced online communication cost.

Table 5 shows detailed benchmark of SP-LUT and ROTL in different n and l settings.

8.4 Boolean LUT benchmark

Given that the communication costs of FLUTE+ and FLUTE are the same during both the preprocessing and online phases (Table 4), our evaluation subsequently centers on runtime performance in LAN setting. We benchmark the online runtime performance of our proposed FLUTE+ framework against the original FLUTE [10] implementation in Figure 13.

FLUTE significantly lags behind FLUTE+ in online runtime, primarily due to its $O(n^2 l)$ computation complexity. When $n = 256$ and $l = 8$, our proposed FLUTE+ is approximately 636× and 167× faster than FLUTE in the LAN and WAN settings respectively. The online runtime of FLUTE+ is almost constant with respect to l because each l-bit element is processed in a single SIMD instructions with AVX2 extension [45].

Table 7 shows detailed benchmark of FLUTE [10] and FLUTE+ (Section 6) in different n and l settings.

Table 5: Detailed benchmark of the Arithmetic LUT with different table size n and elements' bit-length l. We run each LUT in batch size 100 000 and report average communication and runtime.

protocol	table size n	bit-length l	Communication (bit)			LAN (μs)			WAN (μs)		
			total	online	prep.	total	online	prep.	total	online	prep.
SP-LUT [18]	4	8	35.2	34	1.2	6.257	0.057	6.2	8.527	0.787	7.74
		16	67.2	66		6.263	0.063		8.838	1.098	
		32	131.2	130		6.287	0.087		9.646	1.906	
		64	259.2	258		6.340	0.140		11.068	3.328	
	16	8	134.4	132	2.4	6.59	0.128	6.46	9.81	1.952	7.86
		16	262.4	260		6.62	0.155		11.09	3.229	
		32	518.4	516		6.71	0.245		13.95	6.092	
		64	1030.4	1028		6.91	0.448		19.72	11.863	
	64	8	521.6	518	3.6	7.12	0.351	6.77	14.85	6.670	8.18
		16	1033.6	1030		7.23	0.460		20.62	12.437	
		32	2057.6	2054		7.63	0.861		31.17	22.985	
		64	4105.6	4102		8.42	1.653		52.28	44.096	
	256	8	2060.8	2056	4.8	8.82	1.237	7.58	32.60	23.871	8.73
		16	4108.8	4104		9.24	1.656		54.73	46.001	
		32	8204.8	8200		10.83	3.253		98.83	90.101	
		64	16396.8	16392		14.00	6.416		186.85	178.123	
ROTL (Ours) Section 5	4	8	299.2	38	261.2	19.34	0.146	19.19	29.21	3.254	25.96
		16	331.2	70		19.38	0.193		29.57	3.612	
		32	395.2	134		19.41	0.215		30.21	4.254	
		64	523.2	262		19.43	0.240		31.82	5.862	
	16	8	572.4	42	530.4	20.25	0.313	19.94	33.68	3.400	30.28
		16	604.4	74		20.31	0.366		34.04	3.760	
		32	668.4	138		20.33	0.387		34.68	4.400	
		64	796.4	266		20.33	0.390		36.11	5.832	
	64	8	881.6	46	835.6	22.38	0.837	21.54	35.12	3.947	31.17
		16	913.6	78		22.43	0.886		35.48	4.305	
		32	977.6	142		22.45	0.908		36.12	4.946	
		64	1105.6	270		22.51	0.968		37.55	6.377	
	256	8	1334.8	50	1284.8	35.02	3.010	32.01	53.20	6.113	47.09
		16	1366.8	82		35.07	3.058		53.56	6.474	
		32	1430.8	146		35.09	3.076		54.21	7.116	
		64	1558.8	274		35.11	3.100		55.64	8.550	

Table 6: Improvement factor of online runtime of ROTL over SP-LUT.

LAN

n \ l	8	16	32	64
4	0.39	0.32	0.40	0.58
8	0.39	0.39	0.57	0.85
16	0.40	0.40	0.63	1.14
32	0.40	0.50	0.84	1.45
64	0.41	0.52	0.95	1.71
128	0.41	0.53	1.03	1.96
256	0.41	0.54	1.05	2.07

WAN

n \ l	8	16	32	64
4	0.24	0.30	0.44	0.56
8	0.34	0.52	0.78	1.12
16	0.52	0.85	1.38	2.03
32	0.96	1.65	2.73	3.97
64	1.69	2.89	4.64	6.91
128	2.73	4.80	8.21	12.9
256	3.90	7.10	12.7	20.8

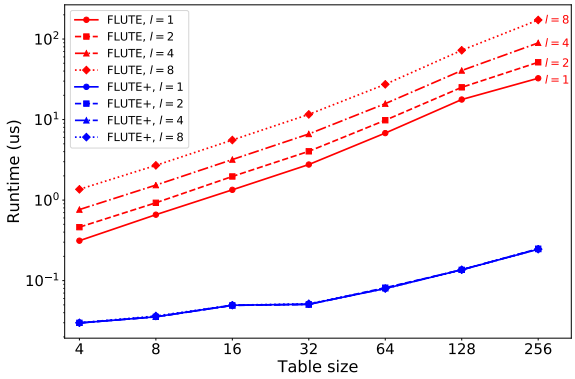


Figure 13: Online runtime comparison between FLUTE and FLUTE+ in LAN setting.

8.5 Evaluation of secure comparison protocol

We compare our secure comparison protocol with the SP-LUT based construction Cheetah [38], FSS DCF based construction [5] and a construction with constant communication round [46]. Table 8 shows the benchmarks in different settings.

Our secure comparison protocol shares the same structure with Cryptflow2 [57], Cheetah [38] and FSS DCF based construction [5], which partition the inputs (l-bits) into m smaller blocks (l'-bits) where $m = \lceil \frac{l}{l'} \rceil$ and use $\mathcal{F}_{\mathsf{LUT}}$ to calculate the `lt` and `eq`. After that, use $\mathcal{F}_{\mathsf{AND}}$ to get the comparison result with tree based combination technique. We give a brief complexity analysis here: 1) for `lt` and `eq`, it needs m $\mathcal{F}_{\mathsf{LUT}}$ instance with $n = 2^{l'}$ and $l = 2$ in one round; 2) for tree based combination, it needs $m-1$ $\mathcal{F}_{\mathsf{AND}}$ instances within $\log m$ rounds. In our implementation, we follow the mainstream setting $l' = 4$, which is widely adopted by [5, 38, 57]. We also evaluate our secure comparison protocol with different l' in Figure 14, The

Table 7: Benchmark of the LUT with different table size n and elements' bit-length l. We run each LUT in batch size 100 000 and report average communication and runtime.

protocol	table size n	bit-length l	Communication (bit)			LAN (μs)			WAN (μs)		
			total	online	prep.	total	online	prep.	total	online	prep.
FLUTE [10]	4	1	9.2	4	5.2	0.29	0.17	0.12	3.10	1.01	2.09
		2				0.43	0.31		3.10	1.01	
		4				0.74	0.62		3.46	1.37	
		8				1.32	1.20		3.81	1.72	
	16	1	65.2	8	57.2	1.11	0.86	0.25	4.41	1.01	3.40
		2				1.79	1.54		4.88	1.48	
		4				3.08	2.83		6.26	2.86	
		8				5.85	5.60		9.09	5.69	
	64	1	308.4	12	296.4	4.42	3.59	0.83	10.57	3.68	6.89
		2				7.53	6.70		13.69	6.80	
		4				14.00	13.17		20.22	13.33	
		8				26.13	25.30		32.67	25.78	
	256	1	1300.4	16	1284.4	35.90	32.59	3.31	53.28	36.03	17.25
		2				61.72	58.41		78.97	61.72	
		4				112.15	108.84		132.27	115.02	
		8				211.93	208.62		236.76	219.51	
FLUTE+ (Ours) Section 6	4	1	9.2	4	5.2	13.93	0.027	13.90	19.28	1.01	18.27
		2				13.93	0.028		19.28	1.01	
		4				13.93	0.029		19.28	1.01	
		8				13.93	0.029		19.28	1.01	
	16	1	65.2	8	57.2	14.90	0.044	14.86	21.79	1.03	20.76
		2				14.91	0.046		21.79	1.03	
		4				14.91	0.048		21.79	1.03	
		8				14.90	0.049		21.79	1.03	
	64	1	308.4	12	296.4	16.10	0.074	16.03	23.64	1.05	22.59
		2				16.11	0.082		23.65	1.06	
		4				16.13	0.097		23.67	1.08	
		8				16.13	0.100		23.68	1.09	
	256	1	1300.4	16	1284.4	47.43	0.230	47.20	57.81	1.19	56.62
		2				47.46	0.256		57.84	1.22	
		4				47.52	0.324		57.93	1.31	
		8				47.53	0.328		57.93	1.31	

Table 8: CMP evaluation with various bit-lengths. we simultaneously run 2^{20} CMP with 8 threads and report average communication and runtime.

bit-length	Protocol	Communication (bit)			LAN time (μs)			WAN time (μs)		
		total	online	prepr.	total	online	prepr.	total	online	prepr.
$l = 32$	SP-LUT based [38]	374	340	34	7.21	2.26	4.95	12.70	6.93	5.77
	FSS DCF based [5]	16 736	64	16672	21.28	8.17	13.11	47.65	28.4	19.25
	constant round [46]	54 216	53 620	596	582.36	1.13	581.23	1879	48	1831
	ROTL based (Ours)	5 546	132	5 414	24.22	0.28	23.94	83.44	2.84	80.60
$l = 64$	SP-LUT based [38]	755	692	63	12.09	2.83	9.26	21.98	12.28	9.70
	FSS DCF based [5]	33 344	128	33 216	42.15	9.25	32.9	144.54	54.9	89.64
	constant round [46]	145 005	1 166	143 839	2098.25	1.13	2096.94	4706	72.91	4633
	ROTL based (Ours)	11 114	276	10 838	48.39	0.57	47.82	165.47	5.86	159.61

experiment result shows that $l' = 4$ achieves the minimal total communication cost and online runtime.

Since a single comparison instance runs in μs, we batch run 2^{20} comparison instances with 8 threads and report the average communication (bit) and runtime (μs). Each experiment is repeated 5 times, and the variance is negligible.

In terms of online performance, we achieve a 5× speedup over the Cheetah [38] in the LAN setting. In the WAN setting, our ROTL based secure comparison protocol is also at least 2× faster than Cheetah. [46] is tailored for constant rounds, our protocol has better performance but needs $O(\log m)$ communication rounds.

8.6 Evaluation of PPML applications

Arithmetic LUT are widely used in PPML applications [29, 30, 36, 54–57, 59], especially for implementing secure non-linear functions like sigmoid, invsqrt and tanh, where the input and output are both arithmetic secure-shared values. We compare our ROTL based PPML protocol with GROTTO[8] [59], which is a state-of-the-art PPML framework based on FSS. We use the same construction proposed in CipherGPT [36], splitting the non-linear function into several segments and using LUT to determine which segment the input

[8]GROTTO: https://github.com/DigitalLibertiesLab/grotto is unavailable, we implemented it following the pseudocode in appendix B of [59].

Table 9: Detailed benchmark of the selected PPML functions. All functions are simultaneous evaluationed in batch size $10,000$ with single thread.

Function	Protocol	Communication (bits)			LAN (us)			WAN (us)		
		total	online	prep.	total	online	prep.	total	online	prep.
sigmoid	GROTTO [59]	25 918	24 318	1 600	126	92	34	697	510	187
	SP-LUT [18]	75 076	7 940	67 136	674	635	39	1 700	1 502	198
	ROTL (Section 5)	10 843	1 624	1 624	130	64	66	724	490	234
invsqrt	GROTTO [59]	25 918	24 318	1 600	154	120	34	598	410	188
	SP-LUT [18]	75 076	7 940	67 136	674	635	39	1 700	1 502	198
	ROTL (Section 5)	10 843	1 624	1 624	130	64	66	724	490	234
tanh	GROTTO [59]	25 918	24 318	1 600	118	84	34	646	460	186
	SP-LUT [18]	75 076	7 940	67 136	674	635	39	1 700	1 502	198
	ROTL (Section 5)	10 843	1 624	1 624	130	64	66	724	490	234
log 10	GROTTO [59]	25 918	24 318	1 600	440	220	220	900	450	450
	SP-LUT [18]	75 076	7 940	67 136	674	635	39	1 700	1 502	198
	ROTL (Section 5)	10 843	1 624	1 624	130	64	66	724	490	234
sqrt	GROTTO [59]	25 918	24 318	1 600	644	610	34	976	790	186
	SP-LUT [18]	75 076	7 940	67 136	674	635	39	1 700	1 502	198
	ROTL (Section 5)	10 843	1 624	1 624	130	64	66	724	490	234

belongs to, then using a low-degree polynomial to approximate the non-linear function in that segment. CipherGPT uses SP-LUT [18] for LUT instances, while we replace it with our proposed ROTL. For fairness, we set $n = 256$ to make sure that the accuracy of our ROTL based PPML protocol is the same as GROTTO. We set the polynomial degree $d = 3$ for all experiments, which aligns with GROTTO.

Table 9 shows that in the LAN setting, our ROTL based PPML protocol is $1.3\times$ - $9.5\times$ faster than GROTTO and around $10\times$ faster SP-LUT based protocol in the online phase. The online runtime efficiency mainly comes from that our proposed ROTL needs no correlation robust hash operation in the online phase. Our ROTL based PPML protocols have the minimal communication cost both in the online phase and the total. SP-LUT based protocols transfer the masked table in the online phase, making the network bandwidth the bottleneck. Moreover, GROTTO uses a relaxed secure model compared to ROTL and SP-LUT, it explicitly requires a trusted helper server in the preprocessing phase to generate FSS keys.

9 Discussion

Although our proposed protocol ROTL is efficient in the online phase, it suffers from high preprocessing communication and computation overhead.

- For preprocessing communication, ROTL involves a factor of $\lambda \log n$ which is $200\times$ expensive than SP-LUT. Table5 shows that ROTL only gains lower total communication when n and l are relatively big.
- For preprocessing computation, ROTL's asymptotic complexity is n^2 while SP-LUT's is $n \log n$, which limits ROTL in scaling up table size.

For preprocessing storage overhead, ROTL requires $n + \log n$ bits storage for each party, while SP-LUT needs $2\lambda \log n$ bits storage for $\mathcal{S}$ and $(\lambda + 1) \log n$ bits storage for C However, in practice, ROTL has lower storage overhead than SP-LUT since the table size n is often less than 256, the security parameter λ is usually set to 128.

Acknowledgments

This work was supported in part by the National Key Research and Development Program of China under Grant 2023YFB2704000, the Fundamental Research Funds for the Central Universities under Grant 226-2025-00004, the Zhejiang Province Pioneer Plan (2024C01074, 2025C01084).

References

[1] Amit Agarwal, Elette Boyle, Niv Gilboa, Yuval Ishai, Mahimna Kelkar, and Yiping Ma. Compressing unit-vector correlations via sparse pseudorandom generators. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology – CRYPTO 2024*, pages 346–383, Cham, 2024. Springer Nature Switzerland.

[2] 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.

[3] 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.

[4] Avrim Blum, Merrick Furst, Michael Kearns, and Richard J. Lipton. Cryptographic primitives based on hard learning problems. In Douglas R. Stinson, editor, *Advances in Cryptology — CRYPTO' 93*, pages 278–291, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[5] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. Function secret sharing for mixed-mode and fixed-point secure computation. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021*, pages 871–900, Cham, 2021. Springer International Publishing.

[6] 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.

[7] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. Cryptology ePrint Archive, Paper 2019/448, 2019.

[8] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015*, pages 337–367, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[9] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. *IACR Cryptol. ePrint Arch.*, page 78, 2012.

[10] 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.

[11] 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.

[12] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, 2017.

[13] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for tfhe. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 377–408, Cham, 2017. Springer International Publishing.

[14] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. Cryptology ePrint Archive, Paper 2018/421, 2018.

[15] Ashish Choudhury and Arpita Patra. *Secure multi-party computation against passive adversaries*. Springer, 2022.

[16] 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.

[17] 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.

[18] 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.

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

[20] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, page 144, 2012.

[21] Radhika Garg, Kang Yang, Jonathan Katz, and Xiao Wang. Scalable Mixed-Mode MPC. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 523–541, 2024.

[22] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford, CA, USA, 2009. AAI3382729.

[23] 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.

[24] Oded Goldreich. *Foundations of cryptography*, volume 2. Cambridge university press, 2001.

[25] Oded Goldreich, Silvio Micali, and Avi Wigderson. *How to Play Any Mental Game, or a Completeness Theorem for Protocols with Honest Majority*, page 307–328. Association for Computing Machinery, New York, NY, USA, 2019.

[26] Charles Gouert, Mehmet Ugurbil, Dimitris Mouris, Miguel de Vega, and Nektarios Georgios Tsoutsos. Ripple: Accelerating programmable bootstraps for FHE with wavelet approximations. Cryptology ePrint Archive, Paper 2024/866, 2024.

[27] 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.

[28] 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.

[29] Kanav Gupta, Neha Jawalkar, Ananta Mukherjee, Nishanth Chandran, Divya Gupta, Ashish Panwar, and Rahul Sharma. Sigma: Secure gpt inference with function secret sharing. *Proceedings on Privacy Enhancing Technologies*, 2024.

[30] Kanav Gupta, Deepak Kumaraswamy, Nishanth Chandran, and Divya Gupta. LLAMA: A low latency math library for secure inference. *Proceedings on Privacy Enhancing Technologies*, 4:274–294, 2022.

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

[32] Meng Hao, Weiran Liu, Liqiang Peng, Hongwei Li, Cong Zhang, Hanxiao Chen, and Tianwei Zhang. Unbalanced circuit-PSI from oblivious key-value retrieval. Cryptology ePrint Archive, Paper 2023/1636, 2023. https://eprint.iacr.org/2023/1636.

[33] David Heath and Vladimir Kolesnikov. One hot garbling. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 574–593, New York, NY, USA, 2021. Association for Computing Machinery.

[34] David Heath, Vladimir Kolesnikov, and Lucien K. L. Ng. Garbled Circuit Lookup Tables with Logarithmic Number of Ciphertexts". In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024*, pages 185–215, Cham, 2024. Springer Nature Switzerland.

[35] 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.

[36] 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.

[37] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012.

[38] 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.

[39] 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.

[40] 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.

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

[42] 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.

[43] Yehuda Lindell. General composition and universal composability in secure multiparty computation. *Journal of cryptology*, 22:395–428, 2009.

[44] 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.

[45] Chris Lomont. Introduction to intel advanced vector extensions. *Intel white paper*, 23(23):1–21, 2011.

[46] Tianpei Lu, Xin Kang, Bingsheng Zhang, Zhuo Ma, Xiaoyuan Zhang, Yang Liu, Kui Ren, and Chun Chen. Efficient 2PC for constant round secure equality testing and comparison. Cryptology ePrint Archive, Paper 2024/949, 2024.

[47] 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.

[48] 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.

[49] Hiraku Morita, Erik Pohle, Kunihiko Sadakane, Peter Scholl, Kazunari Tozawa, and Daniel Tschudi. *MAESTRO: multi-party AES using lookup tables*. USENIX Association, USA, 2025.

[50] 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. https://eprint.iacr.org/2023/1893.

[51] Arpita Patra, Pratik Sarkar, and Ajith Suresh. Fast actively secure ot extension for short secrets. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*.

[52] 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.

[53] 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.
[54] Deevashwer Rathee, Anwesh Bhattacharya, Divya Gupta, Rahul Sharma, and Dawn Song. Secure floating-point training. In *Proceedings of the 32nd USENIX Conference on Security Symposium*, SEC '23, USA, 2023. USENIX Association.
[55] 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, 2022.
[56] 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.
[57] 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.
[58] Phillipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. Distributed vector-ole: Improved constructions and implementation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1055–1072, New York, NY, USA, 2019. Association for Computing Machinery.
[59] Kyle Storrier, Adithya Vadapalli, Allan Lyons, and Ryan Henry. Grotto: Screaming fast (2+1)-pc or $\mathbb{Z}_{2^n}$ via (2,2)-dpfs. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 2143–2157, New York, NY, USA, 2023. Association for Computing Machinery.
[60] Patrick J Van Fleet. *Discrete wavelet transformations: An elementary approach with applications*. John Wiley & Sons, 2019.
[61] 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.
[62] 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.

A Proof of Theorem 1

Proof of Theorem 1.

Correctness. Notice that $\mathbf{c} = (\mathbf{a} \ggg s) - \mathbf{r}$. Then, $\mathcal{S}$'s output in Step 5 is:

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

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

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

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

Then, if C can distinguish the two executions, $\mathcal{A}$ 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 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 $\mathbf{x} - \mathbf{a}$.
- $H_1 = (1^\lambda, n, \mathbf{x}, \widetilde{\mathsf{view}}^{\mathcal{S}}_{\mathsf{Rotate}})$, 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, C could replace $seed^C_s$ with a random seed from the domain. Then, by the pseudorandom property of PRG, C could replace $\mathbf{d}_s$ with $\mathbf{d}'_s \in_\$ \mathbb{Z}^n_{2^l}$ in Step 2.a, which leads to a random $\mathbf{a}$ in Step 2.d. Therefore, $H_1 \approx_c H_0$.
- $H_2 = (1^\lambda, n, \mathbf{x}', \widetilde{\mathsf{view}}^{\mathcal{S}}_{\mathsf{Rotate}})$.
  In H_1, the server's view contains the value $\mathbf{x} - \mathbf{a}'$. In H_2, it contains $\mathbf{x}' - \mathbf{a}'$. Since $\mathbf{a}'$ is a uniformly random sampled from the domain, it acts as a one-time pad, ensuring that $\mathbf{x} - \mathbf{a}'$ and $\mathbf{x}' - \mathbf{a}'$ are information-theoretically indistinguishable. Therefore, $H_2 \approx H_1$, which implies $H_2 \approx_c H_1$.

□

B Proof of Theorem 2

Proof of Theorem 2.

Correctness. We prove the correctness by tracking the important values in the protocol.

- **b (Prep 1):** C defines $\mathbf{b}$ as a one-hot vector with a 1 at index $\langle s \rangle_C$.
- **b′ (Prep 3):** According to the functionality of $\mathbb{F}_{\mathsf{Rotate}}$, $\mathbf{b}'$ is a one-hot vector with a 1 at index $s = (\langle s \rangle_C + \langle s \rangle_{\mathcal{S}}) \bmod n$.
- *sum* **(Prep (4, 5)):** We have $sum = 1$ when $\langle s \rangle_C = 1$ and $\langle s \rangle_{\mathcal{S}} = 0$; $sum = -1$ when $\langle s \rangle_C = 0$ and $\langle s \rangle_{\mathcal{S}} = 1$.
- **b″ (Online 1, 2, 3):** $\mathbf{b}''$ is a one-hot vector with a 1 at index $id = (s + s')$.
- z **(Online 2, 3):** $z = x_{id}$ when $\langle s \rangle_C = 1$ and $\langle s \rangle_{\mathcal{S}} = 0$; $z = -x_{id}$ when $\langle s \rangle_C = 0$ and $\langle s \rangle_{\mathcal{S}} = 1$.
- z' **(Online 4):** In both cases, we have $z' = z \cdot sum = x_{id}$.

Security. We first consider the case where $\mathcal{S}$ is corrupt. We have $\mathcal{S}$ participate in two executions of the protocol, with C inputting $\langle id \rangle_C$ and $\langle id \rangle'_C$ respectively. We show the indistinguishability in a sequence of hybrids:

- H_0 is the real execution with C's input $\langle id \rangle_C$. $\mathcal{S}$'s view is $\mathsf{view}^{\mathcal{S}}_0 = (\langle s \rangle_{\mathcal{S}}, \langle \mathbf{b}' \rangle_{\mathcal{S}}, m_C, \langle z' \rangle_{\mathcal{S}})$, where $m_C = (\langle id \rangle_C - \langle s \rangle_C)$, and $\langle \mathbf{b}' \rangle_{\mathcal{S}}, \langle z' \rangle_{\mathcal{S}}$ are the outputs of $\mathcal{F}_{\mathsf{Rotate}}$ and $\mathcal{F}_{\mathsf{Mult}}$, respectively.
- H_1 is identical to H_0, except we replace all messages received by $\mathcal{S}$ with random values. $\mathcal{S}$'s view is $\mathsf{view}^{\mathcal{S}}_1 = (\langle s \rangle_{\mathcal{S}}, \mathbf{r}_1, r_2, r_3)$, where $\mathbf{r}_1 \in_\$ (\mathbb{Z}_2)^n$, $r_2 \in_\$ \mathbb{Z}_n$, and $r_3 \in_\$ \mathbb{Z}_{2^l}$ are uniformly random sampled from the corresponding domains.
  H_1 is indistinguishable from H_0 due to three properties:
  (1) By the definition of the ideal functionality $\mathcal{F}_{\mathsf{Rotate}}$, the output $\langle \mathbf{b}' \rangle_{\mathcal{S}}$ is computationally indistinguishable from a random sampled vector $\mathbf{r}_1$.

(2) The message $m_C = (\langle id \rangle_C - \langle s \rangle_C)$ is perfectly masked by $\langle s \rangle_C$, which is uniformly random sampled by C in the preprocessing phase, and is independent to $\langle id \rangle_C$. $\langle s \rangle_C$ acts as a one-time pad, making m_C information-theoretically indistinguishable from a uniformly random sampledr_2.

(3) By the definition of the ideal functionality $\mathcal{F}_{\text{Mult}}$, the output $\langle z' \rangle_{\mathcal{S}}$ is computationally indistinguishable from a random sampled value r_3.

Therefore, we have $H_1 \approx_c H_0$.

- H_2: This hybrid is identical to H_1, except it is generated starting from C's input $\langle id' \rangle_C$. $\mathcal{S}$'s view is $\text{view}_2^{\mathcal{S}} = (\langle s \rangle_{\mathcal{S}}, \mathbf{r}_1, r_2, r_3)$. Since $\mathbf{r}_1$, r_2, and r_3 are all uniformly random sampled values, their distribution is independent of C's input. To this end, we have $H_2 \approx_c H_1$.

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

- H_0 is the real execution with $\mathcal{S}$'s input $\langle id \rangle_{\mathcal{S}}$. C's view is $\text{view}_0^C = (\langle s \rangle_C li, \mathbf{b}, \langle \mathbf{b}' \rangle_C, m_{\mathcal{S}}, \langle z' \rangle_C)$, where $m_{\mathcal{S}} = (\langle id \rangle_{\mathcal{S}} - \langle s \rangle_{\mathcal{S}})$, and $\langle \mathbf{b}' \rangle_C, \langle z' \rangle_C$ are the output of $\mathcal{F}_{\text{Rotate}}$ and $\mathcal{F}_{\text{Mult}}$, respectively.
- H_1 is identical to H_0, except we replace all messages received by C with random values. C's view is $\text{view}_1^C = (\langle s \rangle_C, \mathbf{b}, \mathbf{r}_1, r_2, r_3)$, where $\mathbf{r}_1 \in_\$ (\mathbb{Z}_2)^n$, $r_2 \in_\$ \mathbb{Z}_n$, and $r_3 \in_\$ \mathbb{Z}_{2^l}$ are uniformly random sampled from the corresponding domains. We reuse the same proof as in the case where $\mathcal{S}$ is corrupt to prove that $H_1 \approx_c H_0$.
- H_2 is identical to H_1, except it is generated starting from $\mathcal{S}$'s input $\langle id' \rangle'_{\mathcal{S}}$. We reuse the same proof as in the case where $\mathcal{S}$ is corrupt to prove that $H_2 \approx_c H_1$.

□

C Related work

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

Garbled LUTs. Yao's garbled circuits (GC) [62] 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 [35, 47, 52]. This could be considered as a general solution for LUT evaluations. Fairplay [47] supports garbled gates with up to 3-inputs, with their approach generalizing to an arbitrary number of inputs. TASTY [35] supports multi-input garbled gates, incorporating garbled-row reduction. More recently, [52] introduced garbled circuits featuring multi-input/multi-output gates. Heath et al. proposed a novel LUT protocol in [34] based on the One-Hot Garbling (OHG) technique [33].

TFHE programmable bootstrapping based LUT.
TFHE [14] is a fully homomorphic encryption (FHE) scheme, which allows arbitrary computations over binary gates. I. Chillotti et al. [13] proposed *blind rotation* algorithm to accelerate circuit bootstrapping and extend the functionality to enable a LUT evaluation while bootstrapping (i.e., programmable bootstrapping). Ripple [26] optimized programmable bootstrapping LUT with discrete wavelet transform (DWT) [60] to achieve better performance. TFHE programmable bootstrapping based LUT depends on heavy bootstrapping computation and requires transferring homomorphic bootstrapping keys in the size of several gigabytes in advance.

Pseudorandom correlation generator LUT. Agarwal et al. [1] proposed pseudorandom correlation generator (PCG) constructions for unit-vector (UV) correlation. A UV correlation is a vector that contains only one randomly distributed 1, and 0 elsewhere. The vector and 1's position are secret-shared. In [7], Boyle et al. showed that UV correlation can be locally converted to a randomly rotated secret-shared table, then consumed to implement LUT. The efficiency of generating UV correlation is highly dependent on the number of parties. In a two semi-honest party setting, [1] needs around 1 second to generate a UV correlation of length 8 while SOTA protocols run in only tens μs.

Function Secret Sharing based LUT. The Distributed Point Function (DPF) mode of Function Secret Sharing [8] can be used to implement LUT directly. GROTTO [59] proposed a novel FSS DPF-based LUT for the scenario where the public table is composed of sparse continuous values. GROTTO avoids recovering the full DCF tree and perform better than other DPF or DCF based LUT protocols [5, 29, 30].

LUTs w/wo preprocessing. The preprocessing model is widely used in secure multiparty computation (MPC) [3, 17]. 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 phase in terms of communication, interactive rounds, as well as overall runtime.

Ishai et al. [40] 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 representing the table once for every possible input. Dessouky et al. [18] 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 9) is significantly more lightweight.

In the same paper, Dessouky et al. [18] 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* [10]". 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 in secure inference. Cheetah [38] employs LUT to realize the state-of-the-art Π_{CMP}, which is subsequently utilized to implement the ReLU activation function. SecFloat [55] leverages LUT for floating-point computation, striking a commendable balance between efficiency and accuracy. SIRNN [56] pioneers the exploration of employing LUT for computing complex functions such as sigmoid and softmax. Iron [31] applies them to inference transformer model, but it exhibits high costs in communication and computation. CipherGPT [36] introduces an innovative method for computing GELU and softmax, optimizing the utilization of 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 [29] uses function secret-sharing (FSS) to achieve a similar goal with LUT, but this solution relies on a trusted dealer.

D LUT benchmark in different batch size

Both our proposed ROTL and FLUTE+ are implemented based on silent OT. A key feature of silent OT preprocessing (e.g., in Ferret OT [61]) is the generation of a large batch of random OTs (rOTs), often numbering in the millions (10^6). We benchmarked ROTL and FLUTE+ with different batch sizes in Table 10 and found that our approaches exhibit overhead in small-batch scenarios. This is largely because fixed costs—such as serialization, padding, and network latency are amplified when the transferred payload and computational work are minimal. As the batch size increases, this overhead is dramatically reduced, significantly lowering the amortized cost per rOT.

E CMP evaluation with various l'

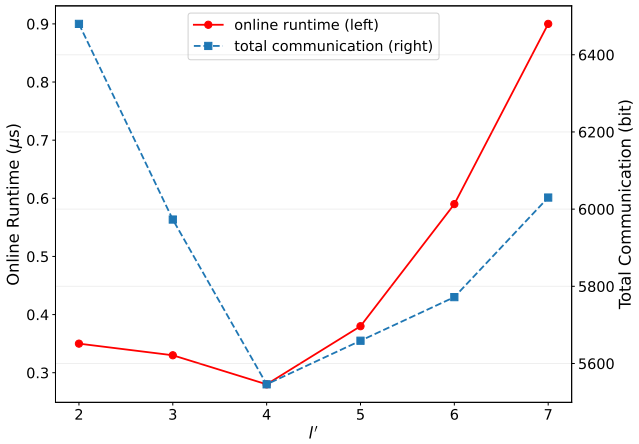
Our secure comparison protocol shares the same structure with Cryptflow2 [57], Cheetah [38] and FSS DCF-based construction [5], which partition the inputs (l-bits) into m smaller blocks (l'-bits) where $m = \lceil \frac{l}{l'} \rceil$ and use $\mathcal{F}_{\mathsf{LUT}}$ calculate the `lt` and `eq`. After that, use $\mathcal{F}_{\mathsf{AND}}$ to get the comparison result with tree-based combination technique. We give a brief complexity analysis here: 1) for `lt` and `eq`, it needs m $\mathcal{F}_{\mathsf{LUT}}$ instance with $n = 2^{l'}$ and $l = 2$ in one round; 2) for tree-based combination, it needs $m-1$ $\mathcal{F}_{\mathsf{AND}}$ instances within $\log m$ rounds. In our implementation, we follow the mainstream setting $l' = 4$, which is widely adopted by [5, 38, 57]. We also evaluate our secure comparison protocol with different l' in Figure 14, The experiment result shows that $l' = 4$ achieves the minimal total communication cost and online runtime.

F Compare ROTL with other LUT

We compare our proposed ROTL with the other two arithmetic LUTs in Table 11.

Scalable MPC [21] proposed an HE [9, 12, 17, 20, 22] (Homomorphic Encryption) based MPC LUT protocol, tailored for a large number of parties (e.g., 32 parties or 64 parties). In our considered two-party scenario, Scalable MPC is less efficient in both runtime and communication cost.

Figure 14: CMP evaluation with $l = 32$ and various l' in LAN setting. we simultaneously run 2^{20} CMP with 8 threads and report average communication and runtime.



MAESTRO [49] is an MPC LUT protocol tailored to perform AES in multi-party (≥ 3) parties. MAESTRO's construction is based on the fact that multiplication of replicated secret-shared value in a multi-party (≥ 3) scenario can be done locally. In our considered two-party scenario, MAESTRO needs to perform lots of expensive secure multiplications, making it impractical to implement.

Table 10: Benchmark of LUT with different table size n and elements' bit-length l in different batch size b. We report the average communication and runtime in the total.

protocol	table size n	bit-length l	Communication (bit)			LAN runtime (μs)			WAN runtime (μs)		
			$b=1$	$b=10^3$	$b=10^5$	$b=1$	$b=10^3$	$b=10^5$	$b=1$	$b=10^3$	$b=10^5$
ROTL Section 5	4	8	$1.6*10^6$	13569.5	299.2	$1.6*10^5$	1312.26	19.34	$2.7*10^5$	2190.37	29.21
	16	16	$1.6*10^6$	13873.5	604.4	$1.6*10^5$	1320.45	20.31	$2.7*10^5$	2197.06	34.04
	64	32	$1.6*10^6$	14245.5	977.6	$1.7*10^5$	1376.65	22.45	$2.8*10^5$	2298.02	36.12
	256	64	$1.6*10^6$	14825.5	1558.8	$1.8*10^5$	1425.48	35.11	$2.8*10^5$	2335.23	55.64
FLUTE+ Section 6	4	1	$1.1*10^6$	8855.65	9.2	$1.0*10^5$	65.20	13.93	$2.0*10^5$	1685.75	19.28
	16	2	$1.1*10^6$	8899.65	65.2	$1.1*10^5$	98.51	14.91	$2.1*10^5$	1759.41	21.79
	64	4	$1.1*10^6$	9087.65	308.4	$1.1*10^5$	104.92	16.13	$2.3*10^5$	1831.21	23.67
	256	8	$1.1*10^6$	9851.65	1300.4	$1.2*10^5$	308.40	47.53	$2.3*10^5$	1854.67	57.93

Table 11: Detailed benchmark of the Arithmetic LUT with different table size n and elements' bit-length l. We run each LUT in batch size 100 000 and report average communication and runtime.

protocol	table size n	bit-length l	Communication (bit)			LAN (μs)			WAN (μs)		
			total	online	prep.	total	online	prep.	total	online	prep.
Scalable MPC [21]	4	≈ 32	8722	4	8718	44.17	0.050	44.12	596.50	1.01	96.40
	16		34890	6	34884	169.53	0.050	169.48	385.92	1.01	384.91
	64		1.39×10^5	12	1.39×10^5	677.98	0.050	677.93	1540.65	1.01	1539.64
	256		5.58×10^5	16	5.58×10^5	2798.46	0.050	2798.41	6151.04	1.01	6150.03
ROTL (Ours) Section 5	4	32	393.2	132	261.2	19.41	0.215	19.19	30.21	4.254	25.96
	16		666.4	136	530.4	20.33	0.387	19.94	34.68	4.400	30.28
	64		975.6	140	835.6	22.45	0.908	21.54	36.12	4.946	31.17
	256		1428.8	144	1284.8	35.09	3.076	32.01	54.21	7.116	47.09
MAESTRO† [49]	16	4	27	16	11	0.3140	0.1289	0.1851	14.83	9.05	5.78
	256	8	38	16	22	0.9250	0.1559	0.7691	21.38	10.68	10.70
ROTL (Ours) Section 5	16	4	570.4	40	530.4	20.25	0.313	19.94	33.68	3.400	30.28
	256	8	1332.8	48	1284.8	35.02	3.010	32.01	53.20	6.113	47.09

† MAESTRO considers 3 participate parties.