Efficient and High-Accuracy Secure Two-Party Protocols for a Class of Functions with Real-number Inputs

Hao Guo[1], Zhaoqian Liu[1], Liqiang Peng[2], Shuaishuai Li[3], Ximing Fu[1,*], Weiran Liu[2], Lin Qu[2]
[1]The Chinese University of Hong Kong, Shenzhen
[2]Alibaba Group
[3]Zhongguancun Laboratory, Beijing, China
E-mails: guohao.g@outlook.com, zhaoqianliu@link.cuhk.edu.cn, plq270998@alibaba-inc.com, liss@zgclab.edu.cn, fuxm07@foxmail.com, weiran.lwr@alibaba-inc.com, xide.ql@taobao.com

Abstract

In two-party secret sharing scheme, values are typically encoded as unsigned integers $\mathsf{uint}(x)$, whereas real-world applications often require computations on signed real numbers $\mathsf{Real}(x)$. To enable secure evaluation of practical functions, it is essential to compute $\mathsf{Real}(x)$ from shared inputs, as protocols take shares as input. At USENIX'25, Guo et al. proposed an efficient method for computing signed integer values $\mathsf{int}(x)$ from shares, which can be extended to compute $\mathsf{Real}(x)$. However, their approach imposes a restrictive input constraint $|x| < \frac{L}{3}$ for $x \in \mathbb{Z}_L$, limiting its applicability in real-world scenarios. In this work, we significantly relax this constraint to $|x| < B$ for any $B \leq \frac{L}{2}$, where $B = \frac{L}{2}$ corresponds to the natural representable range in $x \in \mathbb{Z}_L$. This relaxes the restrictions and enables the computation of $\mathsf{Real}(x)$ with loose or no input constraints. Building upon this foundation, we present a generalized framework for designing secure protocols for a broad class of functions, including integer division ($\lfloor \frac{x}{d} \rfloor$), trigonometric ($\sin(x)$) and exponential ($e^{-x}$) functions. Our experimental evaluation demonstrates that the proposed protocols achieve both high efficiency and high accuracy. Notably, our protocol for evaluating e^{-x} reduces communication costs to approximately 31% of those in SirNN (S&P'21) and Bolt (S&P'24), with runtime speedups of up to 5.53× and 3.09×, respectively. In terms of accuracy, our protocol achieves a maximum ULP error of 1.435, compared to 2.64 for SirNN and 8.681 for Bolt.

1 Introduction

Secure two-party computation (2PC) [28, 31, 32] is a foundational cryptographic primitive that enables two mutually distrustful parties to collaboratively evaluate a public function over their private inputs without revealing any information beyond the function's output. By guaranteeing both correctness and privacy, 2PC has become a foundational technology for privacy-preserving computation, enabling a wide range of real-world applications such as secure data analysis and privacy-preserving machine learning. However, the practical deployment of 2PC is often limited by significant communication overhead, largely driven by expensive cryptographic operations such as those based on public-key primitives. Consequently, the design of efficient and scalable two-party protocols has become a central focus in modern 2PC research.

The Garbled Circuits (GC) method, introduced by Yao [31], was the first general solution proposed for 2PC. While GC provides a robust framework, its high computational and communication costs limit its practicality in real-world scenarios. Recent advances in privacy-preserving machine learning (PPML) [14, 20, 23–25, 29] have spurred the development of customized protocols that prioritize efficiency. Additionally, innovative protocol design principles have emerged to guide the construction of more effective solutions. For instance, SirNN [26] leverages the concept of non-uniform bitwidths to significantly reduce the overhead of various protocols and secure RNN inference tasks. The fitting techniques, such as polynomial fitting and Fourier series fitting, have become standard tools for evaluating non-linear functions, including GELU [13, 25], e^{-x} [26] and sigmoid [17, 22]. The recent work SEAF [11] further advances this area by providing an optimized protocol design method for general non-linear activation functions.

In the two-party secret sharing scheme, an input is encoded as an unsigned fixed-point number $x = \mathsf{uint}(x)$ to perform cryptographic operations, and shared as $x = x_0 + x_1 \bmod L$. However, most real-world functions operate on signed real numbers $\mathsf{Real}(x)$. Consequently, a conversion from $\mathsf{uint}(x)$ to $\mathsf{Real}(x)$ is necessary. Since protocols have access only to the shares x_0 and x_1, $\mathsf{Real}(x)$ needs to be computed from these shares. Guo et al. [12] introduced a new variable $\mathsf{MW}(x)$ to derive the signed value $\mathsf{int}(x)$ from shares via $\mathsf{int}(x) = x_0 + x_1 - \mathsf{MW}(x) \cdot L$. The corresponding signed real value is then $\mathsf{Real}(x) = \frac{\mathsf{int}(x)}{2^f}$, where f denotes the number of fractional bits. However, the approach of Guo et al. has two main limitations. First, they only give an efficient method for computing $\mathsf{MW}(x)$ under the constraint $|x| < \frac{L}{3}$, which is

*Ximing Fu is the corresponding author.

often violated in practice and thus limits applicability. For example, a value $x \in [-50, 50]$ is encoded and shared over ring $\mathbb{Z}_{128}$. In this case, although the upper bound of $|x|$ is smaller than the ring size, we have $|x| > \frac{128}{3}$, and thus the method in [12] fails. Therefore, a new method that supports $|x| < B$ for $B < \frac{L}{2}$ is required. Second, they focus on computing signed integer values, whereas many real-world applications require real number inputs.

To address the limitations in [12], we propose two main improvements and extensions. For $x \in \mathbb{Z}_L$, the geometric method in [12] can compute $\mathsf{MW}(x)$ only under the constraint $|x| < \frac{L}{3}$. In this work, we focus on relaxing this restriction and extend the input range to $|x| < B$ for any $B \leq \frac{L}{2}$. To this end, instead of the AND-based approach in [12], we develop a new comparison-based method that achieves the desired functionality under the relaxed constraint. This extension broadens the range of scenarios in which geometric methods can be applied and improves their overall applicability. When an upper bound B on $|x|$ is known a priori, this relaxed constraint enlarges the set of admissible inputs. In the absence of such prior knowledge, we set $B = \frac{L}{2}$, which is the natural upper bound on $|x|$ for $x \in \mathbb{Z}_L$. Furthermore, we extend the geometric method from signed integers to real numbers. We first compute the $\mathsf{MW}(x)$ as defined in [12], and then represent a signed real number with f-bit precision as $\mathsf{Real}(x) = \frac{x_0 + x_1 - \mathsf{MW}(x) \cdot L}{2^f}$. Then a function $\mathsf{func}(\cdot)$ with $\mathsf{Real}(x)$ as input can be written as $\mathsf{func}(\mathsf{Real}(x)) = \mathsf{func}(\frac{x_0}{2^f} + \frac{x_1}{2^f} + \frac{-\mathsf{MW}(x) \cdot L}{2^f})$. Based on this representation, we present a general protocol design method for a broader class of practical functions, including trigonometric and exponential operations. Our main contributions are summarized as follows:

- Focusing on the constraint limitations in prior work, we relax the constraint for computing $\mathsf{MW}(x)$ from $|x| < \frac{L}{3}$ in prior work to $|x| < B$, where $B \leq \frac{L}{2}$. Notably, when $B = \frac{L}{2}$, the only constraint on x is that $x \in \mathbb{Z}_L$. Then the real number $\mathsf{Real}(x)$ can be computed from shares efficiently. This improvement is enabled by an optimized comparison protocol under input constraints, which is of independent interest.

- We propose a novel method for securely evaluating a class of functions $\mathsf{func}(\cdot)$ which takes $\mathsf{Real}(x)$ as input and satisfies the property $\mathsf{func}(a+b+c) = \sum_i f_i(a) \cdot g_i(b) \cdot h_i(c)$. We instantiate this method to several important real-world functions, including integer division ($\lfloor \frac{x}{d} \rfloor$), trigonometric ($\sin(x)$) and exponential ($e^{-x}$) functions. The implementations demonstrate that our new protocols achieve both low overhead and high accuracy.

1.1 Our results

Comparison protocol with constraint. A comparison protocol takes l-bit x from party P_0 and l-bit y from party P_1 as inputs, and outputs $b = \mathbf{1}\{x < y\}$, with linear communication complexity $O(l)$ [27]. In this work, we focus on comparison protocols with constraints. Specifically, for $x, y \in \mathbb{Z}_L$, where $L = 2^l$, we assume that x is either smaller than y or greater than y by A. Formally, $x - y \in [A, L) \bigcup [-L, 0)$. Under this constraint, we prove that $y < x$ if and only if $\lfloor \frac{y}{A} \rfloor < \lfloor \frac{x}{A} \rfloor$. Consequently, the input length of the comparison protocol is reduced from l to $\lceil \log \lfloor \frac{L}{A} \rfloor \rceil$, and the communication complexity decreases from $O(l)$ (or $O(\lceil \log L \rceil)$) to $O(\lceil \log \lfloor \frac{L}{A} \rfloor \rceil)$.

Computing $\mathsf{MW}(x)$ for $|x| < \frac{L}{2}$. We generalize Guo et al.'s work [12] for computing $\mathsf{MW}(x)$ from constraint $|x| < \frac{L}{3}$ to $|x| < B$ for any $B \leq \frac{L}{2}$. $B = \frac{L}{2}$ implies no additional constraints beyond $x \in \mathbb{Z}_L$. The theoretical communication of our protocol for computing $\mathsf{MW}(x)$ is summarized in Table 1, and some instances are listed in Table 1(b). When $B = 0.9999 \cdot \frac{L}{2}$ (allowing x to take 99.99% of the values in $\mathbb{Z}_L$), only a 14-bit input comparison protocol is required. Even for $B = 0.999999 \cdot \frac{L}{2}$ (covering 99.9999% of values), the input length is 20 bits, achieving $\frac{l}{l^*} = \frac{37}{20} = 1.85$ times improvement compared to the method with natural constraint $B = \frac{L}{2}$, where l-bit comparison protocol is invoked.

Moreover, for the case an l-bit x is shared over larger ring $\mathbb{Z}_{2^{l_r}}$, where $l_r \geq l+1$, we give an efficient method to compute $\mathsf{MW}(x \bmod 2^l, 2^l)$ from $\mathsf{MW}(x, 2^{l_r})$, where the latter can be easily computed. Then, computations can be performed on small ring $\mathbb{Z}_{2^l}$ with low overhead. This technique significantly improves the efficiency for evaluating e^{-x} in subsection 5.3.1.

Table 1: Theoretical communication for our computing $\mathsf{MW}(x)$ protocol $\Pi^{l,l'}_{\mathsf{MW}}$ with constraint $|x| < B$. $\Pi^{l,l'}_{\mathsf{MW}}$ takes shared l-bit x as input and outputs shared l'-bit $\mathsf{MW}(x)$. Parameters are defined as $L = 2^l$, $K = \lfloor \frac{L}{L-2B} \rfloor$. When $B \neq \frac{L}{2}$, let $l^* = \lceil \log \lfloor \frac{L}{L-2B} \rfloor \rceil$; when $B = \frac{L}{2}$, let $l^* = l$.

(a) Theoretical communication for $x \in \mathbb{Z}_L$.

Range of B	Comm.
$[0, \frac{3}{8}L)$	$K \cdot (\lambda + l')$
$[\frac{3}{8}L, \frac{L}{2}]$	$< \lambda(l^*+1) + 14l^* + l'$

(b) Theoretical communication for some examples, where $l = 37$.

Value of B	l^*	Comm.
$0.5 \cdot \frac{L}{2}$	-	165
$0.8 \cdot \frac{L}{2}$	3	< 591
$0.9999 \cdot \frac{L}{2}$	14	< 2153
$0.999999 \cdot \frac{L}{2}$	20	< 3005
$1 \cdot \frac{L}{2}$	37	< 5254

New protocol designs for real-world functions. Using $\mathsf{MW}(x)$ as a foundation, we can compute the signed real number from shares. Further, based on the representation of

Table 2: Theoretical communication for our division, exponential, and trigonometric protocols. The input bitwidth is l, with precision f. C_{MW} denotes the communication of computing $\mathsf{MW}(x)$ protocol. $l_d = \lceil \log d \rceil$, where d is the divisor. Π^1_{rExp} denotes that the protocol takes input with range $\mathsf{Real}(x) \in [0,8)$, and the protocol marked by Π^2_{rExp} works for any $\mathsf{Real}(x) \geq 0$.

Func.	Protocol	Comm.
$\lfloor \frac{x}{d} \rfloor$	Π_{Div}, Sec 5.1	$\lambda(l_d+3)+5l+18l_d+C_{\mathsf{MW}}$
$\sin(x)$	Π_{sin}, Sec 5.2	$69\lambda+1888+C_{\mathsf{MW}}$
e^{-x}	Π^1_{rExp}, Sec 5.3	$28\lambda+2l+4f+897$
	Π^2_{rExp}, Sec 5.3	$\lambda(l+29)+18l+4f+897$

$\mathsf{Real}(x)$, we propose a method for evaluating functions $\mathsf{func}(\cdot)$ with the property $\mathsf{func}(a+b+c) = \sum_{i=0}^{k-1} f_i(a) \cdot g_i(b) \cdot h_i(c)$. This method is applied to several real-world functions, including integer division ($\lfloor \frac{x}{d} \rfloor$), trigonometric ($\sin(x)$) and exponential functions (a^x, e^{-x}). The theoretical communication for these protocols is detailed in Table 2. The experimental results in Section 6.2 demonstrate significant improvements. For our division protocol, we achieve 1.4× to 4.84× improvement over CrypTFlow2 [27]. For the exponential function protocol, compared to SirNN [26] and Bolt [25], we reduce communication costs to 31%, while improving runtime by up to 5.53× and 3.09×, respectively. Furthermore, when applied on the activation function evaluation in PPML, our Softmax protocol, built on the optimized e^{-x} and batch division protocols, achieves 4.59× to 6.2× improvement over Iron [13], and 1.94× to 2.32× improvement over Bolt [25].

Our protocols not only reduce overhead but also achieve higher accuracy. Specifically, the maximum ULP (units in the last place) error of our $\sin(x)$ protocol is approximately 1.3. Our e^{-x} protocol achieves a maximum ULP error of 1.435, outperforming prior works, which reports ULP errors of 2.64 in SirNN and 8.681 in Bolt.

2 Overview

2.1 Computing $\mathsf{Real}(x)$ from Shares

Guo et al. [12] introduced $\mathsf{MW}(x)$ to compute the signed value $\mathsf{int}(x)$ from shares as $\mathsf{int}(x) = x_0 + x_1 - \mathsf{MW}(x) \cdot L$. The $\mathsf{MW}(x)$ is an abbreviation of $\mathsf{MW}(x_0, x_1, L)$ or $\mathsf{MW}(x, L)$, with the constraint $B \leq \frac{L}{2}$, $\mathsf{MW}(x)$ can be computed as:

$$\begin{aligned}\mathsf{MW}(x) =& \mathsf{MW}(x, L) = \mathsf{MW}(x_0, x_1, L) \\ =& \begin{cases} 0, & \text{if } x_0 + x_1 \in [0, B), \\ 1, & \text{if } x_0 + x_1 \in [L-B, L+B), \\ 2, & \text{if } x_0 + x_1 \in [2L-B, 2L). \end{cases}\end{aligned} \tag{1}$$

Using $\mathsf{MW}(x)$, the real value can be computed as

$$\mathsf{Real}(x) = \frac{\mathsf{int}(x)}{2^f} = \frac{x_0 + x_1 - \mathsf{MW}(x) \cdot L}{2^f}, \tag{2}$$

where f denotes the precision. Therefore, the core task for computing $\mathsf{Real}(x)$ is to compute $\mathsf{MW}(x)$. Guo et al. [12] proposed an efficient method to compute $\mathsf{MW}(x)$ under constraint $|x| < \frac{L}{3}$. However, many practical scenarios do not allow this constraint to be satisfied.

Our work focuses on relaxing the constraint to $|x| < B$ for any $B \leq \frac{L}{2}$, and therefore computing $\mathsf{Real}(x)$ with any constraint. In this case, determining whether $\mathsf{MW}(x) = 0$ is equivalent to checking whether $x_0 + x_1 < B$. Let $\alpha = x_0$ and $\beta = B - x_1$, we have that $\mathsf{MW}(x) = 0$ if and only if $\alpha < \beta$, which can be determined using a comparison protocol. Moreover, there is a constraint on the input of this comparison protocol, as $\alpha - \beta = x_0 + x_1 \in [-B, 0) \bigcup [L-2B) \bigcup [2L-2B, 2L-B)$. Therefore, determining $\mathsf{MW}(x) = 0$ reduces to performing a comparison protocol with this specialized constraint. In this work, we propose new protocols for comparison protocol under such constraint in Section 4.1, and demonstrate how to compute $\mathsf{MW}(x)$ by invoking only one comparison protocol in Section 4.2. Finally, $\mathsf{MW}(x)$ can be efficiently computed even if B is very close to $\frac{L}{2}$. For the case $B = \frac{L}{2}$, $\mathsf{MW}(x)$ is computed by invoking only one l-bit comparison protocol.

Furthermore, we consider the scenario in which an l-bit x is shared over the ring $\mathbb{Z}_{2^{l_r}}$, where $l_r \geq l+1$. In this case, $\mathsf{MW}(x, 2^{l_r})$ can be computed efficiently since $|x| < \frac{2^{l_r}}{4}$ [12]. However, protocols are more efficient when executed over the small ring $\mathbb{Z}_{2^l}$ instead of $\mathbb{Z}_{2^{l_r}}$, due to the shorter input lengths involved. To benefit from the advantages of both approaches, we propose a method to derive $\mathsf{MW}(x \bmod 2^l, 2^l)$ from $\mathsf{MW}(x, 2^{l_r})$, as detailed in Section 4.3. Using this method, we first compute $\mathsf{MW}(x, 2^{l_r})$ over the ring $\mathbb{Z}_{2^{l_r}}$, then convert it to $\mathsf{MW}(x \bmod 2^l, 2^l)$, and finally perform protocols over $\mathbb{Z}_{2^l}$. This approach allows us to achieve the high efficiency for computing $\mathsf{MW}(x, 2^{l_r})$ and benefits of performing protocols with shorter input lengths.

2.2 Applications on Real-world Functions

Consider a class of functions $\mathsf{func}(\cdot)$ with the following properties: $\mathsf{func}(\cdot)$ takes a single real number x as input, by dividing x as $x = a+b+c$, $\mathsf{func}(x)$ can be computed as:

$$\mathsf{func}(x) = \mathsf{func}(a+b+c) = \sum_{i=0}^{k-1} f_i(a) \cdot g_i(b) \cdot h_i(c), \tag{3}$$

where f_i, g_i and h_i are public functions. For example, exponential function is an instance of $\mathsf{func}(\cdot)$, as $e^{a+b+c} = e^a \cdot e^b \cdot e^c$. In this case, $k = 1$, and f_0, g_0 and h_0 are all exponential functions. Using the real number representation in Equation 2,

$\mathsf{func}(\cdot)$ with the form of Equation 3 can be computed as:

$$\mathsf{func}(\mathsf{Real}(x)) = \sum_{i=0}^{k-1} f_i(\frac{x_0}{2^f}) \cdot g_i(\frac{x_1}{2^f}) \cdot h_i(\frac{-\mathsf{MW}(x) \cdot L}{2^f}). \quad (4)$$

In this formulation, $f_i(\frac{x_0}{2^f})$ and $g_i(\frac{x_1}{2^f})$ can be computed locally by parties P_0 and P_1, respectively. $h_i(\frac{-\mathsf{MW}(x)\cdot L}{2^f})$ depends on $\mathsf{MW}(x)$, which is a shared value. However, since $\mathsf{MW}(x) \in \{0,1,2\}$, $h_i(\frac{-\mathsf{MW}(x)\cdot L}{2^f})$ can take only three possible values. To compute Equation 4 efficiently, we let parties to compute $w_i^j = f_i(\frac{x_0}{2^f}) \cdot g_i(\frac{x_1}{2^f}) \cdot h_i(\frac{-j\cdot L}{2^f})$ for $j \in \{0,1,2\}$. Here, $h_i(\frac{-j\cdot L}{2^f})$ is public and known to both parties, and therefore multiplying $h_i(\frac{-j\cdot L}{2^f})$ can be computed very efficiently. We store the value w_i^j in a lookup table T, and use $\mathsf{MW}(x)$ as the index to retrieve $w_i = f_i(\frac{x_0}{2^f}) \cdot g_i(\frac{x_1}{2^f}) \cdot h_i(\frac{-\mathsf{MW}(x)\cdot L}{2^f})$. Finally, by summing over all w_i for $i \in \{0,\ldots,k-1\}$, we can get the result of $\mathsf{func}(\mathsf{Real}(x))$. The main overhead of this method involves several multiplications, one computation of $\mathsf{MW}(x)$, and one lookup table protocol. Moreover, the bitwidths and precisions of the components $f_i(\frac{x_0}{2^f})$, $g_i(\frac{x_1}{2^f})$ and $h_i(\frac{-\mathsf{MW}(x)\cdot L}{2^f})$ are independent of the input bitwidth and precision, and can be set arbitrarily. Therefore, highly accurate approximations can be derived by setting high-precision of these components.

If a function has the property of Equation 3, it can be securely evaluated as Equation 4. Examples include truncation, division, trigonometric functions, and exponential functions. Thus, these functions can all be evaluated using this method. The specific protocol designs are listed in Section 5.

3 Preliminaries

Notations. This work considers power-of-two ring $\mathbb{Z}_L$, where $L = 2^l$. The indicator function $\mathbf{1}\{\text{condition}\}$ returns 1 if the condition holds and 0 otherwise. For an unsigned value $x = \mathsf{uint}(x) \in \mathbb{Z}_L$, its signed representation is $\mathsf{int}(x) = \mathsf{uint}(x) - \mathsf{MSB}(x) \cdot L$, where $\mathsf{MSB}(x)$ denotes the most significant bit (MSB). We use $[\![\cdot]\!]$ to denote the two-party additive secret sharing. Specifically, $[\![\cdot]\!]^l$ denotes arithmetic sharing over $\mathbb{Z}_L$, and $[\![\cdot]\!]^B$ denotes Boolean sharing. For $x \in \mathbb{Z}_L$, we define the constraint $|x| < B$ as $x \in [0,B) \cup [L-B,L)$. Note that this differs from the traditional absolute value definition: in our setting, we allow x to take value of $L-B$. Let S be a set, A is a subset of S, and symbol $S \setminus A$ denotes the difference set $\{x | x \in S, x \notin A\}$. $x \gg k$ denotes the right shift k-bit operation. The security parameter λ is set to 128 throughout this work.

Fixed-point representation. To perform secure computation over real numbers, values are first encoded as integers over a ring or field using fixed-point representation. Given a real number $x_{real} \in \mathbb{R}$ represented with l bits, and a fractional precision of f bits, it is encoded as $x = \mathsf{Fix}(x_{real}, l, f) = \lfloor x_{real} \cdot 2^f \rfloor \bmod 2^l$. To recover the original signed real value x_{real} from its unsigned fixed-point representation x, the signed value $\mathsf{int}(x) = x - \mathsf{MSB}(x) \cdot L$ is first computed, then $x_{real} = \mathsf{Real}(x,l,f) = \frac{\mathsf{int}(x)}{2^f}$.

ULP error. Suppose a is a real number and $\widetilde{a}$ is an approximation of a with precision $\widetilde{f}$-bit. This work uses the "units in last place" (ULP) error [10] to measure the approximation error between a and $\widetilde{a}$, which quantifies the number of minimum representable units between the two values. Let the minimum representable units of a and $\widetilde{a}$ be u and $\widetilde{u}$, respectively. They can be expressed as $a = k \cdot u$ and $\widetilde{a} = \widetilde{k} \cdot \widetilde{u}$, where both k and $\widetilde{k}$ are integers. The ULP error is then computed as $\frac{|a-\widetilde{a}|}{\widetilde{u}}$. Previous works [25, 26] typically set $u = \widetilde{u} = 2^{-f}$, resulting in integer-valued ULP errors. In contrast, we set $u = 10^{-6}$ while maintaining $\widetilde{u} = 2^{-\widetilde{f}}$, which yields fractional ULP errors and allows for more precise error measurement.

3.1 Two-party Secure Computation

3.1.1 Two-party additive secret sharing

In two-party additive secret sharing, a secret value x is shared as $x = x_0 + x_1 \bmod L$, where participant P_0 holds x_0 and P_1 holds x_1. This sharing can be represented as $[\![x]\!]^l = ([\![x]\!]_0^l, [\![x]\!]_1^l)$, where $[\![x]\!]_i^l$ is usually abbreviated as x_i for $i \in \{0,1\}$. The state "P_0 and P_1 hold $[\![x]\!]^l$" means that each party P_i holds $[\![x]\!]_i^l$ for $i \in \{0,1\}$. This scheme ensures information-theoretic privacy of the secret x, since each party individually learns nothing about the value of x, even with unbounded computational power.

3.1.2 Oblivious transfer

In 1-out-of-k Oblivious Transfer (OT) [4], denoted as $\binom{k}{1}$-OT$_l$, the sender inputs k messages, each with length l. The receiver inputs a value i and learns the message m_i, while the sender has no output. Utilizing the OT extension technology [18], OT protocols can be efficiently implemented in batches. This work employs the IKNP-style OT extension [16], achieving communication of $2\lambda + kl$ bits in 2 rounds for $\binom{k}{1}$-OT$_l$. Especially, for $\binom{2}{1}$-OT$_l$, the communication is reduced to $\lambda + 2l$. The correlated 1-out-of-2 OT [2] is a variant of OT, denoted as $\binom{2}{1}$-COT$_l$. In this functionality, the sender inputs a value x, and the receiver inputs a choice bit b. As a result, the sender receives a random value r, while the receiver obtains either r or $x + r$, depending on the value of b. The communication cost of $\binom{2}{1}$-COT$_l$ is $\lambda + l$ in 2 rounds.

3.1.3 2PC functionalities

In two-party secret sharing, the expression "P_0 and P_1 invoke $\mathcal{F}_f(x,y)$ to learn $[\![z]\!]^l$" means that P_0 inputs x and P_1 inputs y, and they output $[\![z]\!]^l$ such that $z = f(x,y)$. Especially, "Parties input $[\![a]\!]$" denotes P_0 inputs $[\![a]\!]_0$ and P_1 inputs $[\![a]\!]_1$. This work utilizes the following functionalities:

AND. $\mathcal{F}_{\mathsf{AND}}$ allows P_0 to input a bit a and P_1 to input a bit b, such that both parties obtain a Boolean sharing $[\![c]\!]^B$ satisfying $c = a \wedge b$. This functionality can be implemented using Beaver bit-triples [27], with communication $\lambda + 20$ in 2 rounds.

Bit multiplication (BitMul). The bit multiplication functionality $\mathcal{F}^l_{\mathsf{BitMul}}$ takes the same inputs as $\mathcal{F}_{\mathsf{AND}}$, but outputs an arithmetic sharing $[\![c]\!]^l$ of the result $c = a \wedge b$. It can be implemented using one instance of $\binom{2}{1}$-COT$_l$ [12], with communication $\lambda + l$ in 2 rounds.

Comparison. The comparison functionality $\mathcal{F}^l_{\mathsf{Comp}}$ also known as the Millionaires' functionality, takes l-bit x from P_0 and l-bit y from P_1 as input, and outputs a Boolean sharing $[\![b]\!]^B$ such that $b = \mathsf{Comp}(x,y) = \mathbf{1}\{x < y\}$. CrypTFlow2 [27] implemented an efficient protocol for $\mathcal{F}^l_{\mathsf{Comp}}$, with communication less than $\lambda l + 14l$ in $\log l$ rounds.

DReLU. The DReLU functionality $\mathcal{F}^l_{\mathsf{DReLU}}$ takes an arithmetic sharing $[\![x]\!]^l$ as input, and outputs Boolean sharing $[\![b]\!]^B$ satisfying $b = \mathsf{DReLU}(x) = \mathbf{1}\{x \in [0, 2^{l-1})\}$. $\mathcal{F}^l_{\mathsf{DReLU}}$ can be implemented by invoking $\mathcal{F}^{l-1}_{\mathsf{Comp}}$, with communication less than $\lambda(l-1) + 14(l-1)$ [27].

Lookup table (LUT). The LUT functionality $\mathcal{F}^{m,n}_{\mathsf{LUT}}$ takes a public or shared table T and an arithmetic sharing index $[\![I]\!]^m$ as input, where T contains $M = 2^m$ entries, each of n bits. The functionality returns the arithmetic sharing of the I-th element in T as output. $\mathcal{F}_{\mathsf{LUT}}$ can be implemented by invoking a $\binom{M}{1}$-OT$_n$, with communication $2\lambda + Mn$ bits [6].

Boolean to Arithmetic (B2A). The $\mathcal{F}^l_{\mathsf{B2A}}$ takes $[\![x]\!]^B$ as input, and outputs an arithmetic sharing $[\![y]\!]^l$ such that $x = y$. It can be implemented by invoking a $\binom{2}{1}$-COT$_l$ [27], requiring $\lambda + l$ bits of communication in 2 rounds.

Multiplexer (MUX). The multiplexer functionality $\mathcal{F}^l_{\mathsf{MUX}}$ takes an arithmetic sharing $[\![x]\!]^l$ and a Boolean shares $[\![b]\!]^B$ as input, and outputs $[\![y]\!]^l$ such that $y = x \cdot b$. $\mathcal{F}^l_{\mathsf{MUX}}$ can be implemented by invoking two $\binom{2}{1}$-COT$_l$ [27], with communication $2(\lambda + l)$.

Signed extension (SExt). The signed extension functionality $\mathcal{F}^{l,l'}_{\mathsf{SExt}}$ takes $[\![x]\!]^l$ as input, and outputs $[\![y]\!]^{l'}$ such that $\mathsf{int}(y) = \mathsf{int}(x)$, where $l' > l$. SirNN [26] implements this functionality with communication $\lambda(l+1) + 13l + l'$. Further, for the case the input satisfies the constraint $|x| \leq 2^{l-2}$, Guo et al. propose a more efficient implementation [12], reducing the communication to $\lambda + l' - l$.

Cross term. The functionality $\mathcal{F}^{m,n}_{\mathsf{CrossTerm}}$ takes an m-bit x from P_0 and an n-bit y from P_1, and outputs $[\![z]\!]^{m+n}$ such that $z = xy$. SirNN [26] proposes an implementation of this functionality, with communication $u(\lambda + u/2 + 1/2) + mn$, where $u = \min(m,n)$.

3.1.4 Signed multiplication with non-uniform bitwidths

The signed multiplication with non-uniform bitwidths functionality $\mathcal{F}^{m,n}_{\mathsf{Mul}}$ supports various input formats. In its general form, $\mathcal{F}^{m,n}_{\mathsf{Mul}}$ takes $[\![x]\!]^m$ and $[\![y]\!]^n$ as input, and outputs $[\![z]\!]^{m+n}$ such that $\mathsf{int}(z) = \mathsf{int}(x) \cdot \mathsf{int}(y)$ [26]. This work primarily considers the scenario in which P_0 holds an m-bit x and P_1 holds an n-bit y. In this case, the signed product is expressed as: $\mathsf{int}(z) = \mathsf{int}(x) \cdot \mathsf{int}(y) = (x - 2^m \cdot \mathsf{MSB}(x)) \cdot (y - 2^n \cdot \mathsf{MSB}(y)) = xy - 2^m \cdot \mathsf{MSB}(x) \cdot y - 2^n \cdot \mathsf{MSB}(y)x + 2^{m+n} \cdot \mathsf{MSB}(x) \cdot \mathsf{MSB}(y)$. After applying the modulo operation, the last term is zero. The cross term xy can be computed by invoking a functionality $\mathcal{F}^{m,n}_{\mathsf{CrossTerm}}$. The remaining two terms can be computed using two instances of the multiplexer functionality $\mathcal{F}_{\mathsf{MUX}}$. Further, for the special case where $\mathsf{MSB}(x) = \mathsf{MSB}(y) = 0$, we have $\mathsf{int}(z) = x \cdot y$, and only a single invocation of $\mathcal{F}^{m,n}_{\mathsf{CrossTerm}}$ is required.

For the case where the n-bit y is a public value, both x and y can be (signed) extended to $m+n$ bits using $\mathcal{F}_{\mathsf{SExt}}$, after which the multiplication $xy \bmod 2^{m+n}$ can be computed locally. Since signed extension is free for public values, the communication of this protocol is equivalent to that of a single $\mathcal{F}_{\mathsf{SExt}}$.

3.2 Two-party Geometric Method

For two-party secret sharing $x = x_0 + x_1 \bmod L$, Guo et al. [12] introduce a MSB-Wrap coefficient MW defined as

$$\mathsf{MW}(x_0, x_1, L) = \mathsf{MSB}(x) + \mathsf{Wrap}(x_0, x_1, L), \tag{5}$$

where $\mathsf{MSB}(x) = \mathbf{1}\{x \geq \frac{L}{2}\}$ and $\mathsf{Wrap}(x_0, x_1, L) = \mathbf{1}\{x_0 + x_1 \geq L\}$. Moreover, $\mathsf{MW}(x_0, x_1, L)$ is usually abbreviated as $\mathsf{MW}(x, L)$ or $\mathsf{MW}(x)$. This coefficient $\mathsf{MW}(x)$ enables the computation of the signed integer representation of x as $\mathsf{int}(x) = x_0 + x_1 - \mathsf{MW}(x) \cdot L$. This representation facilitates direct computation of operations such as truncation, expressed as $x \gg k = \lfloor \frac{\mathsf{int}(x)}{2^k} \rfloor$. To compute $\mathsf{MW}(x)$, Guo et al. propose a novel geometric method. Specifically, they interpret the shares x_0 and x_1 as coordinates on a plane, treating secret sharing of x as a point $P(x_0, x_1)$ in the rectangular coordinate system. For $|x| < B$, where $0 \leq B \leq \frac{L}{2}$, they define four sets as:

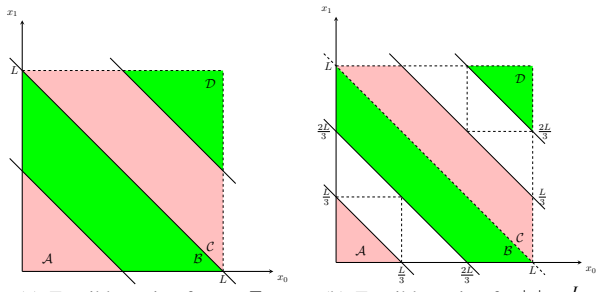
$$\begin{aligned}
&\mathcal{A} : \{(x_0, x_1) | x_0, x_1 \in \mathbb{Z}_L \ , \ 0 \leq x_0 + x_1 < B\}, \\
&\mathcal{B} : \{(x_0, x_1) | x_0, x_1 \in \mathbb{Z}_L \ , \ L - B \leq x_0 + x_1 < L\}, \\
&\mathcal{C} : \{(x_0, x_1) | x_0, x_1 \in \mathbb{Z}_L \ , \ L \leq x_0 + x_1 < L + B\}, \\
&\mathcal{D} : \{(x_0, x_1) | x_0, x_1 \in \mathbb{Z}_L \ , \ 2L - B \leq x_0 + x_1 < 2L\}.
\end{aligned} \tag{6}$$

Then $\mathsf{MSB}(x)$ and $\mathsf{Wrap}(x_0, x_1, L)$ can be computed by determining the area point P falls into, as $\mathsf{MSB}(x) = 0$ if and only if $P \in \mathcal{A} \bigcup \mathcal{C}$, and $\mathsf{Wrap}(x) = 1$ if and only if $P \in \mathcal{C} \bigcup \mathcal{D}$. Therefore, $\mathsf{MW}(x)$ can be written as:

$$\mathsf{MW}(x) = \begin{cases} 0, & \text{if } P \in \mathcal{A}, \\ 1, & \text{if } P \in \mathcal{B} \bigcup \mathcal{C}, \\ 2, & \text{if } P \in \mathcal{D}. \end{cases} \tag{7}$$

as shown in Figure 1(a).

Determining the area in which P falls into is costly for $|x| < \frac{L}{2}$. However, Guo et al. observe that if $|x| < \frac{L}{3}$, then only two AND or BitMul operations are required for determining whether $P \in \mathcal{A}$ or $P \in \mathcal{D}$, as $\mathbf{1}\{P \in \mathcal{A}\} = \mathbf{1}\{x_0 < \frac{L}{3}\} \wedge \mathbf{1}\{x_1 < \frac{L}{3}\}$ and $\mathbf{1}\{P \in \mathcal{D}\} = \mathbf{1}\{x_0 \geq \frac{2L}{3}\} \wedge \mathbf{1}\{x_1 \geq \frac{L}{3}\}$ (as shown in Figure 1(b)). Moreover, for the case $|x| < \frac{L}{4}$, only one BitMul operation is required to compute $\mathsf{MW}(x)$.



(a) Feasible region for $x \in \mathbb{Z}_L$. (b) Feasible region for $|x| < \frac{L}{3}$.

Figure 1: Feasible region of point $P(x_0, x_1)$.

3.3 Threat Model and Security

This work considers secure two-party computation under the threat model adopted by CrypTFlow2 [27]. We assume a static, semi-honest, probabilistic polynomial-time (PPT) adversary that is computationally bounded and may corrupt at most one of the two parties. In the semi-honest model, the corrupted party follows the protocol honestly but attempts to learn private input information from the other party. We formalize security using the simulation paradigm [5, 19]. For a given function f, we compare two settings: the real interaction and the ideal interaction. In the real interaction, the two parties execute the protocol in the presence of an adversary and an external environment $\mathcal{Z}$. In the ideal interaction, the parties send their inputs to a trusted third party, referred to as the functionality $\mathcal{F}$, which computes f and returns the shares of output to parties. A protocol is secure if, for every real-world adversary, there exists a simulator $\mathcal{S}$ in the ideal model such that no environment $\mathcal{Z}$ can distinguish between the real and ideal interactions. If such a simulator exists, the real interaction reveals no information about the private inputs, and the protocol is secure. In practice, a protocol Π is often built from multiple sub-protocols, which is described as hybrid model in CrypTFlow2. The simulation of Π follows the structure of a real interaction, except that each sub-protocol is replaced by an invocation of the corresponding ideal functionality. When Π invokes a functionality $\mathcal{F}$, the protocol is said to operate in the "$\mathcal{F}$-hybrid model". The security of Π then depends on the security of the underlying sub-protocols it calls.

4 Compute $\mathsf{MW}(x)$ with Arbitrary Constraint

According to the definition of $\mathsf{MW}(x)$ in Equation 5 or Equation 1, computing $\mathsf{MW}(x)$ involves the invocation of comparison protocols. Moreover, under the constraint $|x| < B$, there exists an additional constraint on the inputs to these comparison protocols. In this section, we first give a method for performing comparison protocol under special constraint, and subsequently apply this approach to compute $\mathsf{MW}(x)$ for $|x| < B$, with an arbitrary $B \leq \frac{L}{2}$. Furthermore, for the case that l-bit x is shared over l_r-bit ring, where $l_r \geq l+1$, we show how to compute $\mathsf{MW}(x \bmod 2^l, 2^l)$ from $\mathsf{MW}(x, 2^{l_r})$ efficiently.

4.1 Comparison Protocol with Constraint

In general comparison protocol, two participants P_0 and P_1 hold inputs $x, y \in \mathbb{Z}_L$, respectively. The goal is to compute a comparison function defined as: $\mathsf{Comp}(x, y) = \mathbf{1}\{x < y\}$. This work considers a constrained scenario where a specific relationship exists between x and y. Specifically, we assume that x is either less than y or greater than y by a public value A, where $A \in (0, L)$. Formally, the relationship between x and y can be expressed as $x - y \in [A, L) \bigcup [-L, 0)$. Under this constraint, we present Lemma 1, which demonstrates that the comparison protocol for inputs y and x can be reduced to the comparison protocol with inputs $\lfloor \frac{y}{A} \rfloor$ and $\lfloor \frac{x}{A} \rfloor$. Consequently, the input length of the comparison protocol can be reduced from l to $\lceil \log \lfloor \frac{L}{A} \rfloor \rceil$, and the communication complexity is reduced from $O(l)$ to $O(\lceil \log \lfloor \frac{L}{A} \rfloor \rceil)$. The proof of Lemma 1 is provided in Appendix A.1.

Lemma 1. *Let $x, y \in \mathbb{Z}_L$ satisfy $x - y \in [A, L) \bigcup [-L, 0)$, where $0 < A < L$. Then, $\mathsf{Comp}(y, x) = \mathsf{Comp}(\lfloor \frac{y}{A} \rfloor, \lfloor \frac{x}{A} \rfloor)$.*

In two-party secret sharing, the comparison protocol is usually used to compute Wrap function, which can be written as $\mathsf{Wrap}(x_0, x_1, L) = \mathsf{Comp}(L - x_0, x_1)$. Therefore, for the case the range of $x_0 + x_1$ is constrained as $x_0 + x_1 \in [0, L) \bigcup [L + A, 2L)$, we give Lemma 2 to compute $\mathsf{Wrap}(x_0, x_1, L)$, whose proof is provided in Appendix A.2.

Lemma 2. *For $x = x_0 + x_1 \bmod L$, if $x_0 + x_1 \in [0, L) \bigcup [L + A, 2L)$, where $0 < A < L$, then $\mathsf{Wrap}(x_0, x_1, L) = \mathsf{Comp}(\lfloor \frac{L - x_0}{A} \rfloor, \lfloor \frac{x_1}{A} \rfloor)$.*

By applying Lemmas 1 and 2, the Wrap protocol can be realized by a comparison protocol with input length $\lceil \log \lfloor \frac{L}{A} \rfloor \rceil$ instead of l, leading to a reduction in communication complexity from $O(l)$ to $O(\lceil \log \lfloor \frac{L}{A} \rfloor \rceil)$. However, the communication of comparison protocol reported in CrypTFlow2 [27] is an estimated value, which makes it unsuitable for scenarios with very small input lengths. In the following, we present a detailed analysis of the communication complexity for comparison protocols with small input lengths.

4.1.1 Comparison protocol with very small input length

The comparison protocol in CrypTFlow2 [27] relies on $\binom{2^m}{1}$-OT and AND operations, where m is typically set to 4. For small input lengths (e.g. $l \leq 4$), the comparison protocol can be implemented using a single $\binom{2^l}{1}$-OT, with communication $2\lambda + 2^l$. Additionally, CrypTFlow2's comparison protocol outputs a Boolean sharing $[\![b]\!]^B$, and an extra B2A protocol is required to convert it to arithmetic sharing $[\![b]\!]^{l'}$. Therefore, the total communication of the comparison protocol with arithmetic sharing output is $3\lambda + 2^l + l'$ when $l \leq 4$.

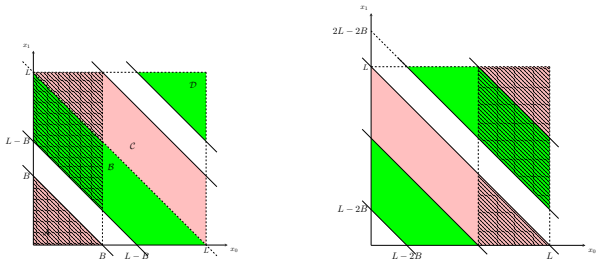
For very small inputs, an AND-based method can also be used to implement the comparison protocol. Moreover, we can use BitMul instead of AND to directly output arithmetic sharing. Specifically, suppose P_0 and P_1 hold inputs x and y, where $x, y \in \mathbb{Z}_n$. The comparison function can be computed as: $\mathsf{Comp}(x,y) = \sum_{i=0}^{n-2} \mathbf{1}\{x = i\} \wedge \mathbf{1}\{y > i\}$. This requires $n-1$ BitMul protocols, directly producing $[\![b]\!]^{l'}$, with a total communication cost $(n-1)\cdot(\lambda + l')$ in 2 rounds. When comparing these two methods, we find that the AND-based method is more efficient when $n \leq 3$. Notably, the method in [12] can be regarded as an instance of the AND-based comparison protocol.

4.2 Computing $\mathsf{MW}(x)$ with Constraint

From the definition of $\mathsf{MW}(x)$ in Equation 5, two comparison protocols are required to compute $\mathsf{MSB}(x)$ and $\mathsf{Wrap}(x_0, x_1, L)$, respectively. However, using the idea of Theorem 2 in [12], we can reduce the number of comparison protocols to one. Specifically, consider the case where $|x| < B$ for $B \leq \frac{L}{2}$. The feasible region of the point $P(x_0, x_1)$ is illustrated in Figure 2(a). By shifting this feasible region to the left by B, we define a new point $P^*(x_0^*, x_1)$ where $x_0^* = x_0 - B \bmod L$. The feasible region for P^* is depicted in Figure 2(b). We introduce a new variable $M^* = \mathbf{1}\{x_0^* + x_1 \geq 2L - 2B\}$, which indicates whether P^* lies within the upper-right triangular region in Figure 2(b). Using this definition, $\mathsf{MW}(x)$ can be computed based on M^*, as formalized in Theorem 1. The proof of this theorem is provided in Appendix A.3.

Theorem 1. *Let $x = x_0 + x_1 \bmod L$ and $|x| < B$, where $B \leq \frac{L}{2}$. Define $x_0^* = x_0 - B \bmod L$, $M^* = \mathbf{1}\{x_0^* + x_1 \geq 2L - 2B\}$ and $\Delta = \mathbf{1}\{x_0 \geq B\}$. Then $\mathsf{MW}(x) = M^* + \Delta$.*

Notably, Theorem 2 in [12] is a special case of Theorem 1 in this work, where they set $B = \frac{L}{4}$. Furthermore, Theorem 1 also holds for $B = \frac{L}{2}$, enabling its application to any $x \in \mathbb{Z}_L$ without constraint. In Theorem 1, Δ can be computed locally by P_0. Thus, the primary challenge in computing $\mathsf{MW}(x)$ lies in determining $M^* = \mathbf{1}\{x_0^* + x_1 \geq 2L - 2B\}$. For $|x| < B$, the value of $x_0^* + x_1$ is constrained to the range $[L-2B, L) \bigcup [2L-2B, 2L)$, as shown in Figure 2(b). If $B = \frac{L}{2}$, then $\mathsf{Wrap}(x_0^*, x_1, L) = \mathsf{Comp}(x_0^*, L - x_1) \oplus 1$. For the case $B \neq \frac{L}{2}$, from Lemma 2, we can derive that $M^* = \mathsf{Wrap}(x_0^*, x_1, L) = \mathsf{Comp}(\lfloor \frac{L - x_0^*}{L-2B} \rfloor, \lfloor \frac{x_1}{L-2B} \rfloor)$. Under the constraint $|x| < B$, defining $l^* = \lceil \log \lfloor \frac{L}{L-2B} \rfloor \rceil$ for $B \neq \frac{L}{2}$, and $l^* = l$ when $B = \frac{L}{2}$, comparison protocol with l^*-bit input is required for computing M^* and $\mathsf{MW}(x)$. The detailed procedure for computing $\mathsf{MW}(x)$ is outlined in Algorithm 1.



(a) Feasible region of $P(x_0, x_1)$. (b) Feasible region of $P^*(x_0^*, x_1)$.

Figure 2: Feasible region of $P(x_0, x_1)$ and $P^*(x_0^*, x_1)$.

In this Algorithm 1, the comparison protocol can be implemented using either the AND-based method or CrypTFlow2's method, depending on which achieves the best efficiency. Specifically, let the maximum input value of comparison protocol for computing M^* be defined as $K = \lfloor \frac{L}{L-2B} \rfloor$. The AND-based method is more efficient for computing M^* when $K \leq 3$, which implies $B < \frac{3L}{8}$. Under this constraint, computing M^* requires only K BitMul protocols. For the case $B \geq \frac{3L}{8}$, the CrypTFlow2's comparison protocol is invoked.

Correctness and security. The correctness of Π_{MW} is guaranteed by Theorem 1. The security of Π_{MW} is ensured by the security of the underlying protocols for functionalities $\mathcal{F}_{\mathsf{BitMul}}$, $\mathcal{F}_{\mathsf{Comp}}$, and $\mathcal{F}_{\mathsf{B2A}}$.

Complexity. For $B < \frac{3L}{8}$, Π_{MW} invokes K instances of the $\mathcal{F}_{\mathsf{BitMul}}^{l'}$, resulting in a total communication cost of $K \cdot (\lambda + l')$ in two rounds. For $B \geq \frac{3L}{8}$, the protocol requires one $\mathcal{F}_{\mathsf{Comp}}^{l^*}$ and one $\mathcal{F}_{\mathsf{B2A}}^{l'}$, with a total communication cost of $(l^*+1)\lambda + 14l^* + l'$ in $2 + \log l^*$ rounds.

4.3 Computing MW via Ring Conversion

Suppose a protocol Π_f consisting of two parts: computing $\mathsf{MW}(x)$ and other operations, with their respective communication complexity denoted as C_{MW} and C_{other}, where C_{other} scales linearly with the input length. Consider the scenario that an l-bit value x is shared over ring $\mathbb{Z}_{2^{l_r}}$, where $l_r \geq l + 1$. To evaluate $f(x)$, the straightforward method is to perform Π_f over the ring $\mathbb{Z}_{2^{l_r}}$, resulting $C_{\mathsf{MW}} \approx \lambda$ as $|x| < 2^{l_r - 2}$, while C_{other} is $O(l_r)$. One can also first compute $z_i = x_i \bmod 2^l$ for $i \in \{0,1\}$ and $z = z_0 + z_1 \bmod 2^l$, and perform Π_f with $[\![z]\!]^l$ as input. This reduces C_{other} to $O(l)$, while C_{MW} increases to $O(l)$, as the l-bit comparison protocol is required to compute $\mathsf{MW}(x, 2^l)$. To benefit from the advantages of both methods, we propose a technique to compute $\mathsf{MW}(z, 2^l)$ from

Algorithm 1: Computing $\mathsf{MW}(x)$ for $|x| < B$, $\Pi^{l,l'}_{\mathsf{MW}}$:

Input: P_0 and P_1 hold $[\![x]\!]^l$ satisfying $|x| < B$, where $B \leq \frac{L}{2}$.

Output: P_0 and P_1 output $[\![\mathsf{MW}(x)]\!]^{l'}$.

Let $L = 2^l$, $K = \lfloor \frac{L}{L-2B} \rfloor$, $l^* = \lceil \log \lfloor \frac{L}{L-2B} \rfloor \rceil$ if $B < \frac{L}{2}$, and $l^* = l$ if $B = \frac{L}{2}$.

P_0 computes $\delta = \mathbf{1}\{x_0 \geq B\}$ and $x_0^* = x_0 - B \bmod L$.

`// CrypTFlow2-based Comparison Protocol`

if $B \geq \frac{3L}{8}$ **then**

  if $B = \frac{L}{2}$ **then**

    P_0 and P_1 invoke $\mathcal{F}^{l^*}_{\mathsf{Comp}}(x_0^*, L - x_1)$ and learn $[\![temp]\!]^B$.

    P_0 and P_1 compute $[\![M^*]\!]^B = [\![temp]\!]^B \oplus 1$.

  end

  else

    P_0 and P_1 invoke $\mathcal{F}^{l^*}_{\mathsf{Comp}}(\lfloor \frac{L-x_0^*}{L-2B} \rfloor, \lfloor \frac{x_1}{L-2B} \rfloor)$ and learn $[\![M^*]\!]^B$.

  end

  P_0 and P_1 invoke $\mathcal{F}^{l'}_{\mathsf{B2A}}([\![M^*]\!]^B)$ and learn $[\![M^*]\!]^{l'}$.

end

`// AND-based Comparison Protocol`

if $B < \frac{3L}{8}$ **then**

  for $i \in \{0, ..., K-1\}$ **do**

    P_0 sets $a_i = \mathbf{1}\{\lfloor \frac{L-x_0^*}{L-2B} \rfloor = i\}$ and P_1 sets $b_i = \mathbf{1}\{\lfloor \frac{x_1}{L-2B} \rfloor > i\}$.

    P_0 and P_1 invoke $\mathcal{F}^{l'}_{\mathsf{BitMul}}(a_i, b_i)$ to learn $[\![c_i]\!]^{l'}$.

  end

  P_0 and P_1 compute $[\![M^*]\!]^{l'} = \sum_{i=0}^{K-1} [\![c_i]\!]^{l'}$.

end

P_0 and P_1 compute and output $[\![MW]\!]^{l'} = [\![M^*]\!]^{l'} + \delta$.

$\mathsf{MW}(x, 2^{l_r})$ with low overhead, reducing the total communication complexity of Π_f to $\lambda + O(l)$.

4.3.1 Computing $\mathsf{MW}(z, 2^l)$ from $\mathsf{MW}(y, 2^{l+1})$

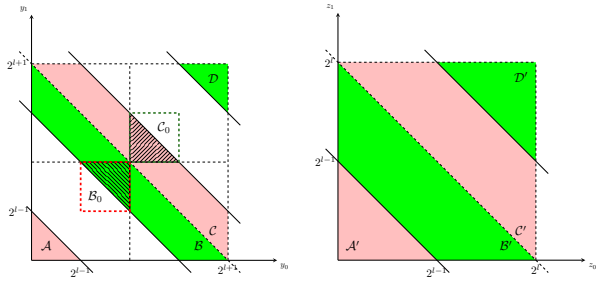
For $x = x_0 + x_1 \bmod 2^{l_r}$, where $|x| < 2^{l-1}$, let $y_i = x_i \bmod 2^{l+1}$ for $i \in \{0,1\}$ and $y = y_0 + y_1 \bmod 2^{l+1}$. Then $\mathsf{int}(x) = \mathsf{int}(y)$, $|y| < 2^{l-1}$, and $\mathsf{MW}(y, 2^{l+1})$ can be easily computed as $|y| < \frac{2^{l+1}}{4}$. Let $z_i = x_i \bmod 2^l$ for $i \in \{0,1\}$ and $z = z_0 + z_1 \bmod 2^l$, we show how to compute $\mathsf{MW}(z, 2^l)$ from $\mathsf{MW}(y, 2^{l+1})$. Using the geometric method in [12], in addition to the feasible regions $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$ defined in Equation 6, we introduce two additional regions $\mathcal{B}_0$ and $\mathcal{C}_0$, for $x \in \mathbb{Z}_N$, defined as:

$$\mathcal{B}_0 : \{(x_0, x_1) | (x_0, x_1) \in \mathcal{B} \ , \ x_0 < \frac{N}{2}, x_1 < \frac{N}{2}\},$$
$$\mathcal{C}_0 : \{(x_0, x_1) | (x_0, x_1) \in \mathcal{C} \ , \ x_0 \geq \frac{N}{2}, x_1 \geq \frac{N}{2}\}.$$

Then, the feasible region of point $P_y(y_0, y_1)$ ($y \in \mathbb{Z}_{2L}$) is illustrated in Figure 3(a), where in this figure the value of N is $2L$. By mapping point $P(x_0, x_1)$ to point $P_z(z_0, z_1)$, where $z_i = x_i \bmod 2^l$ for $i \in \{0,1\}$, the feasible region of point P_z is shown in Figure 3(b). This region is divided into four subregions, denoted as $\mathcal{A}'$, $\mathcal{B}'$, $\mathcal{C}'$ and $\mathcal{D}'$. From these two figures, the relationship between points $P_y(y_0, y_1)$ and $P_z(z_0, z_1)$ can be summarized as follows:

- If $P_y \in \mathcal{A} \bigcup \mathcal{C}_0$, then $P_z \in \mathcal{A}'$.
- If $P_y \in \mathcal{D} \bigcup \mathcal{B}_0$, then $P_z \in \mathcal{D}'$.
- If $P_y \in (\mathcal{B} \bigcup \mathcal{C}) \setminus (\mathcal{B}_0 \bigcup \mathcal{C}_0)$, then $P_z \in \mathcal{B}' \bigcup \mathcal{C}'$.

Thus, we can conclude that $\mathsf{MW}(y, 2^{l+1}) = \mathsf{MW}(z, 2^l)$, except in cases where $P_y \in \mathcal{C}_0 \bigcup \mathcal{B}_0$. Furthermore, whether this exception occurs can be determined using two AND operations. This allows us to compute $\mathsf{MW}(z, 2^l)$ from $\mathsf{MW}(y, 2^{l+1})$. The following Lemma 3 formalizes this proposition, with its proof provided in Appendix A.4.



(a) Feasible region of $P_y(y_0, y_1)$. (b) Feasible region of $P_z(z_0, z_1)$.

Figure 3: Feasible region of $P_y(y_0, y_1)$ and $P_z(z_0, z_1)$, where $y \in \mathbb{Z}_{2^{l+1}}$ and $z \in \mathbb{Z}_{2^l}$.

Lemma 3. *Suppose* $y = y_0 + y_1 \bmod 2^{l+1}$ *satisfying* $|y| < 2^{l-1}$. *For* $i \in \{0,1\}$, *let* $z_i = y_i \bmod 2^l$ *and* $z = z_0 + z_1 \bmod 2^l$. *Define:*

$$\begin{cases} a = \mathbf{1}\{y_0 \in [2^l, 2^l + 2^{l-1})\} \wedge \mathbf{1}\{y_1 \in [2^l, 2^l + 2^{l-1})\}, \\ b = \mathbf{1}\{y_0 \in [2^l - 2^{l-1}, 2^l)\} \wedge \mathbf{1}\{y_1 \in [2^l - 2^{l-1}, 2^l)\}. \end{cases} \tag{8}$$

Then $\mathsf{MW}(z, 2^l) = \mathsf{MW}(y, 2^{l+1}) - a + b$.

4.3.2 Reducing the number of AND operations to one

For given $\mathsf{MW}(y, 2^{l+1})$, only $a - b$ is required to compute $\mathsf{MW}(z, 2^l)$, which typically involves two additional AND operations. Now, we demonstrate how to compute $a - b$ using only a single AND operation. From Equation 8 we have that $a = \mathbf{1}\{P_y \in \mathcal{C}_0\}$, and $b = \mathbf{1}\{P_y \in \mathcal{B}_0\}$. For point P_y in Figure 3(a), we map $P_y(y_0, y_1)$ to $\hat{P}_y(\hat{y}_0, \hat{y}_1)$, where $\hat{y}_i = y_i + 2^l \bmod 2^{l+1}$, and let $\hat{y} = \hat{y}_0 + \hat{y}_1 \bmod 2^{l+1}$. Then in Figure 3(a), we can deduce that:

- $P_y \in \mathcal{C}_0$ (or equivalently, $a = 1$) if and only if $\hat{P}_y \in \mathcal{A}$ (or equivalently, $\mathsf{MW}(\hat{y}, 2^{l+1}) = 0$).
- $P_y \in \mathcal{B}_0$ (or equivalently, $b = 1$) if and only if $\hat{P}_y \in \mathcal{D}$ (or equivalently, $\mathsf{MW}(\hat{y}, 2^{l+1}) = 2$).

Therefore, computing $a - b$ reduces to evaluating $\mathsf{MW}(\hat{y}, 2^{l+1})$, which requires only a single AND operation as $|y| < 2^{l-1}$. The formal explanation is provided in Theorem 2, with its proof detailed in Appendix A.5.

Theorem 2. *Suppose* $y = y_0 + y_1 \bmod 2^{l+1}$, *where* $|y| < 2^{l-1}$, *and a, b are defined as Equation 8. Let* $\hat{y}_i = y_i + 2^l \bmod 2^{l+1}$ *for* $i \in \{0,1\}$, $\hat{y}_0^* = \hat{y}_0 - 2^{l-1} \bmod 2^{l+1}$, *and* $\delta = \mathbf{1}\{\hat{y}_0 \geq 2^{l-1}\}$. *Define* $\hat{M}^* = \mathbf{1}\{\hat{y}_0^* \geq 2^l\} \wedge \mathbf{1}\{\hat{y}_1 \geq 2^l\}$. *Then* $a - b = 1 - \delta - \hat{M}^*$.

Using Lemma 3 and Theorem 2, $\mathsf{MW}(z, 2^l)$ can be computed from $\mathsf{MW}(y, 2^{l+1})$ with only one additional AND operation. The details for computing $\mathsf{MW}(z, 2^l)$ from $x \in \mathbb{Z}_{2^{l_r}}$ is shown in Algorithm 2.

Algorithm 2: Computing $\mathsf{MW}(z, 2^l)$ from $x \in \mathbb{Z}_{2^{l_r}}$, $\Pi^{l_r,l'}_{\mathsf{MW}_{\mathsf{conv}}}$:

Input: P_0 and P_1 hold $[\![x]\!]^{l_r}$, where $|x| < 2^{l-1}$ and $l_r \geq l+1$.
Output: P_0 and P_1 output $[\![\mathsf{MW}_z]\!]^{l'}$, where $\mathsf{MW}_z = \mathsf{MW}(z, 2^l)$. z is shared over $\mathbb{Z}_{2^l}$ and $[\![z]\!]^l_i = [\![x]\!]^{l_r}_i \bmod 2^l$ for $i \in \{0,1\}$.

For $i \in \{0,1\}$, P_i computes $[\![y]\!]^{l+1}_i = x_i \bmod 2^{l+1}$.
`// Compute MW(y)`
P_0 and P_1 invoke $\mathcal{F}^{l+1,l'}_{\mathsf{MW}}([\![y]\!]^{l+1})$ to learn $[\![\mathsf{MW}_y]\!]^{l'}$.
`// Compute a - b`
For $i \in \{0,1\}$, P_i computes $\hat{y}_i = [\![y]\!]^{l+1}_i + 2^l \bmod 2^{l+1}$.
P_0 computes $\hat{y}_0^* = \hat{y}_0 - 2^{l-1} \bmod 2^{l+1}$, and $\delta = \mathbf{1}\{\hat{y}_0 \geq 2^{l-1}\}$.
P_0 and P_1 invoke $\mathcal{F}^{l'}_{\mathsf{BitMul}}(\mathbf{1}\{\hat{y}_0^* \geq 2^l\}, \mathbf{1}\{\hat{y}_1 \geq 2^l\})$ and learn $[\![\hat{M}^*]\!]^{l'}$.
P_0 and P_1 compute $[\![c]\!]^{l'} = 1 - \delta - [\![\hat{M}^*]\!]^{l'}$.
`// Compute MW(z)`
P_0 and P_1 compute $[\![\mathsf{MW}_z]\!]^{l'} = [\![\mathsf{MW}_y]\!]^{l'} - [\![c]\!]^{l'}$.
P_0 and P_1 output $[\![\mathsf{MW}_z]\!]^{l'}$.

Correctness and security. The correctness of $\Pi^{l_r,l'}_{\mathsf{MW}_{\mathsf{conv}}}$ is ensured by Lemma 3 and Theorem 2. The security comes from the security of protocols for $\mathcal{F}_{\mathsf{MW}}$ and $\mathcal{F}_{\mathsf{BitMul}}$.

Complexity. The communication of $\Pi^{l_r,l'}_{\mathsf{MW}_{\mathsf{conv}}}$ arises from the invocation of $\mathcal{F}^{l+1,l'}_{\mathsf{MW}}$ and $\mathcal{F}^{l'}_{\mathsf{BitMul}}$. In $\mathcal{F}^{l+1,l'}_{\mathsf{MW}}$, the constraint is $|y| < \frac{2^{l+1}}{4}$, resulting $K = 1$ in Algorithm 1, and only one Π_{BitMul} is required, with communication $\lambda + l'$. Therefore, the total communication of $\Pi^{l_r,l'}_{\mathsf{MW}_{\mathsf{conv}}}$ is $2(\lambda + l')$.

4.4 Discussion of the Constraint

Our Π_{MW} requires $|x| < B$, and for the case the constraint is not satisfied, an error may occur. Although the error e is a two-bit value, $\mathsf{MW}(x)$ is typically used as $\mathsf{MW}(x) \cdot L$, resulting in a large error term $e \cdot L$. Therefore, the bound on $|x|$ must be carefully evaluated to avoid the large error. There are also some PPML or secure computation works that assume the values in the computation task are bounded by a given bound [9, 15, 23, 24, 33, 34]. Although some works [34] have demonstrated the plausibility of this assumption, we believe that it does not always hold, and once an error occurs, the entire inference or secure computation may be corrupted. Therefore, in this work we use alternative methods to ensure that the constraint is satisfied.

The prior work [12] has proposed some methods to satisfy the constraint $|x| < \frac{L}{4}$. We further propose two additional methods to satisfy this constraint. The first method computes the bound of values in computation tasks in advance. Although these values are secret-shared, their range can be computed in plaintext, since the structure of the computation is usually public. Thus, we can determine a suitable bound on $|x|$. Taking the computation of a mean $\frac{\sum_{i=0}^{14}[\![a_i]\!]^{14}}{15}$ as an example, where each a_i is a 10-bit value shared over $\mathbb{Z}_{2^{14}}$. Let $[\![b]\!]^{14} = \sum_{i=0}^{14}[\![a_i]\!]^{14}$, then we have $\mathsf{int}(b) \in [-7680, 7680)$ and $|b| < 0.9375 \cdot \frac{2^{14}}{2}$. Our method can then be applied to compute $\mathsf{MW}(b)$ and the division $\frac{[\![b]\!]^{14}}{15}$ with high efficiency.

Another approach uses the ring conversion method. Specifically, for $x \in \mathbb{Z}_L$ we encode and share it over $\mathbb{Z}_{2L}$, so that the constraint becomes $|x| < \frac{2L}{4}$. This method was also proposed in [12]; however, sharing over a larger ring introduces additional overhead. To address this, we use the method in Section 4.3 to reduce this overhead. In particular, we use ring conversion to share values over a larger ring to satisfy the constraint, while computing MW over a smaller ring. This approach enables efficient computation of MW without introducing excessive additional cost. It is especially suitable when the effective range of x is relatively small. For example, in the e^{-x} evaluation protocol in Section 5.3.1, we assume $x \in [0, 8)$, and $\mathsf{MW}(x)$ can be computed using $\Pi_{\mathsf{MW}_{\mathsf{conv}}}$ without any additional constraints.

5 Application on Real-World Functions

This work focuses on evaluating a function $\mathsf{func}(\cdot)$ with the form of Equation 3. Once $\mathsf{MW}(x)$ is obtained, Equation 4 can be used to compute $\mathsf{func}(\cdot)$, following the approach outlined in Section 2.2. The evaluation process proceeds as follows:

1. For $i \in \{0, ..., k-1\}$, P_0 and P_1 locally compute $f_i(\frac{x_0}{2^f})$, and $g_i(\frac{x_1}{2^f})$, respectively. Then they compute public functions $h_i(\frac{-j \cdot L}{2^f})$ for $j \in \{0,1,2\}$.

2. For $i \in \{0,...,k-1\}$, parties invoke multiplication protocol to compute $z_i = f_i(\frac{x_0}{2^f}) \cdot g_i(\frac{x_1}{2^f})$.
3. For $i \in \{0,...,k-1\}$, and $j \in \{0,1,2\}$ parties compute $t_i^j = z_i \cdot h_i(\frac{-j \cdot L}{2^f})$, and store them into a lookup table T_i.
4. P_0 and P_1 invoke $\mathcal{F}_{\mathsf{MW}}$ to get $\mathsf{MW}(x)$.
5. For $i \in \{0,...,k-1\}$, P_0 and P_1 query the lookup table T_i with index $\mathsf{MW}(x)$ to get t_i^{MW}.
6. P_0 and P_1 locally compute and output $\sum_i^k t_i$.

This method requires only a limited number of multiplications, a computation of MW, and a lookup table protocol. In the following subsections, we demonstrate the applicability of this approach to real-world functions.

5.1 Division and Truncation Functions

The division function can be expressed in the form of Equation 3 as $\frac{a+b+c}{d} = \frac{a}{d} \cdot 1 \cdot 1 + 1 \cdot \frac{b}{d} \cdot 1 + 1 \cdot 1 \cdot \frac{c}{d}$. This work adopts the definition of signed integer division in CrypTFlow2 [27]. Given a shared input x, and a public divisor d encoded on ring $\mathbb{Z}_L$, the integer division function can be written as:

$$\begin{aligned} \mathsf{Div}(x,d) &= \lfloor \frac{\mathsf{int}(x)}{d} \rfloor = \lfloor \frac{x_0 + x_1 - \mathsf{MW}(x) \cdot L}{d} \rfloor \\ &= \lfloor \frac{x_0}{d} \rfloor + \lfloor \frac{x_1}{d} \rfloor - \mathsf{MW}(x) \cdot \lfloor \frac{L}{d} \rfloor + e, \end{aligned} \tag{9}$$

where e is a small error resulting from the floor operations. To realize faithful division protocol, the error e should be computed and eliminated. We reformulate Equation 9 as:

$$\mathsf{Div}(x,d) = \lfloor \frac{\mathsf{int}(x)}{d} \rfloor = \lfloor \frac{x_0}{d} \rfloor + \lfloor \frac{x_1 - \mathsf{MW}(x) \cdot L}{d} \rfloor + \varepsilon, \tag{10}$$

where $\varepsilon \in \{0,1\}$ is a one-bit error. According to Lemma 2 from [12], ε can be computed as:

$$\varepsilon = \mathbf{1}\{x_0 \bmod d + (x_1 - \mathsf{MW}(x) \cdot L \bmod d) \geq d\}. \tag{11}$$

The remaining challenge lies in computing the terms $\lfloor \frac{x_1 - \mathsf{MW}(x) \cdot L}{d} \rfloor$ and $(x_1 - \mathsf{MW}(x) \cdot L) \mod d$, which can be computed using lookup table (LUT) technology. Specifically, we construct two lookup tables, T^d and T^ε, where $T^d[j] = \lfloor \frac{x_1 - j \cdot L}{d} \rfloor$ and $T^\varepsilon[j] = (x_1 - j \cdot L) \mod d$ for $j \in \{0,1,2\}$. These tables are indexed by $\mathsf{MW}(x)$, allowing parties to retrieve the required values efficiently. Finally, the parties invoke the DReLU protocol to compute the error ε. The details of the division protocol are shown in Algorithm 3.

Correctness and security. The correctness of Π_{Div} is ensured by Equation 10 and Equation 11. The security relies on the security of protocols for $\mathcal{F}_{\mathsf{MW}}$, $\mathcal{F}_{\mathsf{LUT}}$, $\mathcal{F}_{\mathsf{DReLU}}$, and $\mathcal{F}_{\mathsf{B2A}}$.

Complexity. The two $\mathcal{F}_{\mathsf{LUT}}$ invoked by Π_{Div} can be combined as one with 4 entries, each with $l + l_d + 1$ bits. Let C_{MW} denotes the communication of protocol for $\mathcal{F}_{\mathsf{MW}}$, the total communication of $\Pi_{\mathsf{Div}}^{l,d}$ is $\lambda(l_d+3) + 5l + 18l_d + C_{\mathsf{MW}}$.

Algorithm 3: Faithful division protocol, $\Pi_{\mathsf{Div}}^{l,d}$:

Input: P_0 and P_1 hold $[\![x]\!]^l$ and a public d, satisfying $|x| < B$ where $B \leq \frac{L}{2}$.

Output: P_0 and P_1 output $[\![\lfloor \frac{\mathsf{int}(x)}{d} \rfloor]\!]^l$.

Let $l_d = \lceil \log d \rceil$.
for $j \in \{0,1,2\}$ **do**
  P_0 sets $[\![T^d]\!]_0^l[j] = 0$ and $[\![T^\varepsilon]\!]_0^{l_d+1}[j] = 0$.
  P_1 sets $[\![T^d]\!]_1^l[j] = \lfloor \frac{x_1 - j \cdot L}{d} \rfloor$ and $[\![T^\varepsilon]\!]_1^{l_d+1}[j] = (x_1 - j \cdot L) \mod d$.
end
P_0 and P_1 invoke $\mathcal{F}_{\mathsf{MW}}^{l,2}([\![x]\!]^l)$ and learn $[\![\mathsf{MW}]\!]^2$.
P_0 and P_1 invoke $\mathcal{F}_{\mathsf{LUT}}([\![T^d]\!]^l, [\![\mathsf{MW}]\!]^2)$ to learn $[\![X_1]\!]^l$.
P_0 and P_1 invoke $\mathcal{F}_{\mathsf{LUT}}([\![T^\varepsilon]\!]^{l_d+1}, [\![\mathsf{MW}]\!]^2)$ to learn $[\![I_\varepsilon]\!]^{l_d+1}$.
P_0 and P_1 compute $[\![temp]\!]^{l_d+1} = (x_0 \mod d) + [\![I_\varepsilon]\!]^{l_d+1} - d$.
P_0 and P_1 invoke $\mathcal{F}_{\mathsf{DReLU}}^{l_d+1}([\![temp]\!]^{l_d+1})$ to learn $[\![\varepsilon]\!]^B$.
P_0 and P_1 invoke $\mathcal{F}_{\mathsf{B2A}}^{l}([\![\varepsilon]\!]^B)$ to learn $[\![\varepsilon]\!]^l$.
P_0 computes and outputs $\lfloor \frac{x_0}{d} \rfloor + [\![X_1]\!]_0^l + [\![\varepsilon]\!]_0^l$, and P_1 computes and outputs $[\![X_1]\!]_1^l + [\![\varepsilon]\!]_1^l$.

5.1.1 Truncation protocol

The truncation function can be viewed as a special case of the division function, where the divisor is a power of two, i.e., $d = 2^k$. In this case, Equation 9 can be rewritten as $\mathsf{Trun}(x,k) = \mathsf{Div}(x,2^k) = \lfloor \frac{x_0}{d} \rfloor + \lfloor \frac{x_1}{d} \rfloor - \mathsf{MW}(x) \cdot 2^{l-k} + e'$, where $e' = \mathbf{1}\{x_0 \bmod 2^k + x_1 \bmod 2^k \geq 2^k\}$. Accordingly, the truncation protocol can be implemented by invoking the computing $\mathsf{MW}(x)$ protocol, a DReLU (or comparison) protocol, and a B2A protocol. The total communication of the truncation protocol is $\lambda(k+1) + l + 13k + C_{\mathsf{MW}}$.

5.2 Trigonometric Function

We consider the computation of the trigonometric function $\sin(x)$, where $\cos(x)$ can be computed using a similar method. The function $\sin(x)$ fits the form of Equation 3 by applying the sum formula for sine, which expresses $\sin(a+b+c)$ as:

$$\begin{aligned} \sin(a+b+c) = &\sin(a)\cos(b)\cos(c) + \cos(a)\sin(b)\cos(c) \\ &+ \cos(a)\cos(b)\sin(c) - \sin(a)\sin(b)\sin(c). \end{aligned}$$

Therefore, the value of $\sin(\mathsf{Real}(x))$ can be computed using the following formulation:

$$
\begin{aligned}
\sin(\mathsf{Real}(x)) &= \sin(\frac{x_0 + x_1 - \mathsf{MW}(x)\cdot L}{2^f}) \\
&= \sin(\frac{x_0}{2^f})\cos(\frac{x_1}{2^f})\cos(-\mathsf{MW}(x)\cdot\frac{L}{2^f}) \\
&+ \cos(\frac{x_0}{2^f})\sin(\frac{x_1}{2^f})\cos(-\mathsf{MW}(x)\cdot\frac{L}{2^f}) \\
&+ \cos(\frac{x_0}{2^f})\cos(\frac{x_1}{2^f})\sin(-\mathsf{MW}(x)\cdot\frac{L}{2^f}) \\
&- \sin(\frac{x_0}{2^f})\sin(\frac{x_1}{2^f})\sin(-\mathsf{MW}(x)\cdot\frac{L}{2^f}).
\end{aligned} \tag{12}
$$

The computation of $\sin(\mathsf{Real}(x))$ involves one invocation of the $\mathsf{MW}(x)$, several multiplications, and two LUT protocols. The details are listed in Appendix B.

5.3 Exponential Function

The exponential function is a typical instance of the function in Equation 4. Let $a > 0$ be a public base and let x be an exponent with bitwidth l and precision f. Then, $a^{\mathsf{Real}(x)}$ can be computed as:

$$
a^{\mathsf{Real}(x)} = a^{\frac{x_0}{2^f}} \cdot a^{\frac{x_1}{2^f}} \cdot a^{\frac{-\mathsf{MW}(x)\cdot L}{2^f}}. \tag{13}
$$

However, when l is much larger than f, the bitwidth required to represent $a^{\frac{x_i}{2^f}}$ is large, increasing significantly overhead for multiplying $a^{\frac{x_0}{2^f}}$ and $a^{\frac{x_1}{2^f}}$. To mitigate this issue, we set $l = f + \alpha$ and restrict the input range to $[-2^{\alpha-1}, 2^{\alpha-1})$. Let $A_i = a^{\frac{x_i}{2^f}}$ for $i \in \{0,1\}$. Then, the range of A_i is 1 to $a^{\frac{L}{2^f}} = a^{2^\alpha}$. The integer part of A_i can be represented using $\mu = \lceil \frac{L}{2^f} \log a \rceil + 1$ bits, where the additional one bit is used for the sign. We also assign a precision of f_A bits to A_0 and A_1, resulting in a total bitwidth of $l_A = \mu + f_A$. In practice, we set $\alpha = 3$ and $f_A = 10$, which achieves both high efficiency and accuracy. For computing $a^{\frac{-\mathsf{MW}(x)\cdot L}{2^f}}$, which can take values of $1, a^{-2^\alpha}$ and $a^{-2^{\alpha+1}}$, a higher precision f_M is required to ensure accuracy, since the latter two values are very close to zero. Fortunately, f_M has minimal impact on the total communication cost of the protocol, and can be set large. For the setting $\alpha = 3$, we set $f_M = 32$, and use a total bitwidth of $l_M = f_M + 2$, including one bit for the integer part and one for the sign.

Based on these parameter settings, we present our exponential protocol $\Pi_{\mathsf{Exp}}^{l,f,l',f'}$ in Algorithm 4. This protocol assumes the input x is shared over $l = f + 3$ bits ring. However, it can also be applied when x is shared over a larger ring $\mathbb{Z}_{2^{l_r}}$, as long as $|x| < 2^{f+2}$. In such case, given an input $[\![x]\!]^{l_r}$, we compute $[\![x]\!]_i^{f+3} = [\![x]\!]_i^{l_r} \mod 2^{f+3}$ for $i \in \{0,1\}$ to get $[\![x]\!]^{f+3}$, and use $[\![x]\!]^{f+3}$ as input to $\Pi_{\mathsf{Exp}}^{l,f,l',f'}$ to compute $a^{\mathsf{Real}(x)}$.

Correctness and security. The correctness of Π_{Exp} is ensured by Equation 13. The security of Π_{Exp} relies on the security of protocols for $\mathcal{F}_{\mathsf{Mul}}$, $\mathcal{F}_{\mathsf{MW}}$, and $\mathcal{F}_{\mathsf{LUT}}$.

Algorithm 4: Computing a^x, $\Pi_{\mathsf{Exp}}^{l,f,l',f'}$:

Input: P_0 and P_1 hold $[\![x]\!]^l$ with precision f where $l = f + 3$, and a public positive a.
Output: P_0 and P_1 output $[\![Exp]\!]^{l'}$ with precision f', such that $\mathsf{Real}(Exp) = a^{\mathsf{Real}(x)}$.

1 Let $\mu = \lceil 2^{l-f} \cdot \log a \rceil + 1$, $f_A = 10$, and $l_A = \mu + f_A$.
2 Let $f_M = 32$ and $l_M = f_M + 2$.
3 For $i \in \{0,1\}$, P_i locally computes $A_i = a^{\frac{x_i}{2^f}}$, and encode A_i as $\hat{A}_i = \mathsf{Fix}(A_i, l_A, f_A)$.
4 P_0 and P_1 invoke $\mathcal{F}_{\mathsf{Mul}}^{l_A,l_A}(\hat{A}_0, \hat{A}_1)$ and learn $[\![B]\!]^{2l_A}$.
5 **for** $j \in \{0,1,2\}$ **do**
6 Parties computes $M_j = a^{-\frac{j\cdot L}{2^f}}$, and encodes it as $\hat{M}_j = \mathsf{Fix}(M_j, l_M, f_M)$.
7 P_0 and P_1 invoke $\mathcal{F}_{\mathsf{Mul}}^{2l_A,l_M}([\![B]\!]^{2l_A}, \hat{M}_j)$ and learn $[\![Exp_j]\!]^{2l_A+l_M}$.
8 For $i \in \{0,1\}$, P_i computes $[\![Exp_j]\!]_i^{2\mu+2+f'} = [\![Exp_j]\!]_i^{2l_A+l_M} \gg (2f_A + f_M - f') \bmod 2^{2\mu+2+f'}$.
9 P_0 and P_1 set $[\![T]\!][j]^{2\mu+2+f'} = [\![Exp_j]\!]^{2\mu+2+f'}$.
10 **end**
11 P_0 and P_1 invoke $\mathcal{F}_{\mathsf{MW}}^{l,2}([\![x]\!]^l)$ to learn $[\![\mathsf{MW}]\!]^2$.
12 P_0 and P_1 invoke $\mathcal{F}_{\mathsf{LUT}}([\![T]\!]^{2\mu+2+f'}, [\![\mathsf{MW}]\!]^2)$ and learn $[\![Exp]\!]^{2\mu+2+f'}$.
13 For $i \in \{0,1\}$, P_i computes $[\![Exp]\!]_i^{l'} = [\![Exp]\!]_i^{2\mu+2+f'} \bmod 2^{l'}$.
14 P_0 and P_1 output $[\![Exp]\!]^{l'}$.

Complexity. Due to the properties of exponential function, both the A_0 and A_1 are positive, and the $\mathcal{F}_{\mathsf{Mul}}^{l_A,l_A}$ in line 4 can be implemented using $\mathcal{F}_{\mathsf{CrossTerm}}^{l_A,l_A}$. Furthermore, the three instances of $\mathcal{F}_{\mathsf{Mul}}^{2l_A,f_M+2}$ (in line 7) can be realized by a single invocation of $\mathcal{F}_{\mathsf{SExt}}^{2l_A,2l_A+f_M+2}$, as $\hat{M}_j$ is public and the value of B is unchanged for $j \in \{0,1,2\}$. Moreover, since $|\hat{A}_0 \cdot \hat{A}_1| < 2^{l_A-2}$, the $\mathcal{F}_{\mathsf{SExt}}^{2l_A,2l_A+f_M+2}$ can be efficiently implemented using the method proposed in [12], which incurs small overhead. Let C_{MW} denote the communication cost of $\mathcal{F}_{\mathsf{MW}}^{l,2}$. The total communication cost of the exponential protocol is approximately: $\lambda(l_A+3) + \frac{3}{2}l_A^2 + 8\mu + 4f' + C_{\mathsf{MW}}$, where $\mu = \lceil 8 \cdot \log a \rceil + 1$ and $l_A = \mu + 10$. For the case where x is shared over a large ring $\mathbb{Z}_{2^{l_r}}$ for $l_r > f + 3$, $\Pi_{\mathsf{MW}_{\mathsf{conv}}}$ is invoked to compute MW, with communication $2(\lambda+2)$. In this scenario, the total communication of Π_{Exp} is reduced to $\lambda(l_A+5) + \frac{3}{2}l_A^2 + 8\mu + 4f'$.

5.3.1 Evaluating e^{-x}

Considering the function e^{-x}, for $x \geq 0$, we first divide the range of x into two intervals: $x \in [0,8) \bigcup [8,+\infty)$. When x belongs to the second interval, we have $e^{-x} \leq e^{-8} \leq 3.35 \times 10^{-4}$, which is negligible. Consequently, e^{-x} is approximated

as 0 for $x \geq 8$, resulting in the ULP error less than 1.37. For $x \in [0,8)$, let $y = -x+4$, then $e^{-x} = e^{y} \cdot e^{-4}$ and $y \in (-4,4]$. However, the input range of Π_{Exp} is $[-4,4)$. To address this discrepancy, we set $y' = y - \tau$, where $\tau = 2^{-f}$. Then, e^{-x} is approximated as $e^{y'} \cdot e^{-4}$, and $y' \in [-4,4)$ can serve as input of Π_{Exp}. The details of our evaluating e^{-x} protocol are shown in Algorithm 5. In this algorithm, the DReLU protocol is employed to determine whether x lies in interval $[0,8)$ or $[8,+\infty)$. For $x \in [0,8)$, Π_{Exp} is utilized to compute e^{-x}. Moreover, if $l = f+4$, then the range of x is $[-8,8)$. In this case, it is unnecessary to check whether $x > 8$, allowing the DReLU protocol to be omitted, and only a single invocation of Π_{Exp} is required.

Algorithm 5: Computing e^{-x} for $x \geq 0$, $\Pi^{l,f}_{\mathsf{rExp}}$:

Input: P_0 and P_1 hold $[\![x]\!]^l$ with precision f, where $\mathsf{int}(x) \geq 0$ and $l \geq f+4$.
Output: P_0 and P_1 output $[\![rExp]\!]^l$ with precision f, such that $\mathsf{Real}(rExp) = e^{-\mathsf{Real}(x)}$.

1 P_0 and P_1 compute $[\![z]\!]^{f+3} = -[\![x]\!]^l + 4 \cdot 2^f - 1 \mod 2^{f+3}$.
2 P_0 and P_1 invoke $\mathcal{F}^{f+3,f,l,f}_{\mathsf{Exp}}([\![z]\!]^{f+3}_i)$ to learn $[\![sExp]\!]^{l'}$.
3 P_0 and P_1 compute $[\![Exp]\!]^l = [\![sExp]\!]^l \cdot \mathsf{Fix}(e^{-4}, l, f)$.
4 P_0 and P_1 invoke $\mathcal{F}^{l}_{\mathsf{DReLU}}(-[\![x]\!]^l + 8 \cdot 2^f)$ to learn $[\![b]\!]^B$.
5 P_0 and P_1 invoke $\mathcal{F}_{\mathsf{MUX}}([\![Exp]\!]^l, [\![b]\!]^B)$ to learn $[\![rExp]\!]^l$.
6 P_0 and P_1 output $[\![rExp]\!]^l$.

Correctness and security. The correctness of Π_{rExp} comes from that for $x \in [0,8)$, we have $e^{-x} \approx e^{-x+4-2^{-f}} \cdot e^{-4}$. The security relies on the security of protocols for $\mathcal{F}_{\mathsf{Exp}}$, $\mathcal{F}_{\mathsf{DReLU}}$, and $\mathcal{F}_{\mathsf{MUX}}$.

Complexity. $\Pi^{l,f}_{\mathsf{rExp}}$ needs one call for each of $\mathcal{F}^{l}_{\mathsf{DReLU}}$, $\mathcal{F}^{l,f,l,f}_{\mathsf{Exp}}$, and $\mathcal{F}^{l}_{\mathsf{MUX}}$. Since $l > f+3$, the $\mathcal{F}^{l,f,l,f}_{\mathsf{Exp}}$ invokes $\Pi_{\mathsf{MW}_{\mathsf{conv}}}$ to compute MW. Moreover, in $\mathcal{F}^{l,f,l,f}_{\mathsf{Exp}}$, $\mu = 13$ and $l_A = 23$. Therefore, the total communication of $\Pi^{l,f}_{\mathsf{rExp}}$ is approximately $\lambda(l+29) + 18l + 4f + 897$. Moreover, if $l = f+4$, then the overhead of $\Pi^{l,f}_{\mathsf{rExp}}$ is the same as $\Pi^{l,f,l,f}_{\mathsf{Exp}}$, approximated as $28\lambda + 4f + 2l + 897$.

5.3.2 Evaluating Softmax

The Softmax function is a commonly used activation function in machine learning, particularly in multi-class classification tasks. For an input vector $\vec{z} = \{z_0, z_1, ..., z_{n-1}\}$, Softmax is defined as:

$$\mathsf{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=0}^{n-1} e^{z_j}}. \tag{14}$$

In both PPML and plaintext implementations, each z_i is normalized as $z_i^* = z_i - z_{max}$, where z_{max} is the maximum value in $\vec{z}$. This normalization ensures that the inputs to the exponential function are non-positive while preserving the correctness of Softmax. Consequently, the protocol Π_{rExp} can be used to compute $e^{z_i^*}$. The division required for Softmax can be implemented using SirNN's method [26]. First, the reciprocal $D = \frac{1}{\sum_{i=0}^{n-1} e^{z_i}}$ is computed. Then $\mathsf{Softmax}(z_i)$ is obtained by computing $e^{z_i} \cdot D$ for $i \in \{0,1,...,n-1\}$. Since D is computed only once for the entire vector, its amortized overhead is negligible. In the multiplication step, the same D is multiplied by n different values. Therefore, these multiplications are reformulated as a matrix multiplication $[D] \cdot [z_0, z_1, ..., z_{n-1}]$, and computed by invoking the matrix multiplication protocol in SirNN [26]. Finally, we can get $\mathsf{Softmax}(z_i)$ for $i \in \{0,1,...,n-1\}$.

6 Experiments

Experimental setup. Our experiments were conducted on a server equipped with Intel(R) Xeon(R) Platinum processors operating at 2.5 GHz, featuring 8 logical CPUs and 16 GB of memory. To simulate various network conditions, we used Linux Traffic Control (tc), emulating a LAN environment with 1 Gbps bandwidth and 0.8 ms RTT latency, and a WAN environment with 100 Mbps bandwidth and 80 ms RTT latency. All experiments, except those in Section 6.2.5, were built upon the SCI library [1], which employs IKNP-style OT protocols. The experiments in Section 6.2.5 were conducted with silent-OT [3] to evaluate performance under different underlying OT extension technique.

6.1 Computing MW(x)

Prior work [12] proposed an efficient method for computing $\mathsf{MW}(x)$ under the constraint $|x| < \frac{L}{3}$. In this work, we extend this approach by relaxing the upper bound to $|x| < B$ for any $B \leq \frac{L}{2}$. We conduct experiments on our computing $\mathsf{MW}(x)$ protocol Π_{MW}, under varying constraints of $|x| < B$, where $B \in \{0.5 \cdot \frac{L}{2}, 0.8 \cdot \frac{L}{2}, 0.9999 \cdot \frac{L}{2}, 0.999999 \cdot \frac{L}{2}, \frac{L}{2}\}$. These settings demonstrate the performance of Π_{MW} across different bounds. The experimental results are shown in Table 3. For $B = 0.5 \cdot \frac{L}{2}$, our Π_{MW} is the same as the protocol in [12], achieving maximum efficiency. However, for $B > \frac{L}{3}$, the method in [12] fails, while our Π_{MW} remains effective. If it is known that $|x| < 0.9999 \cdot \frac{L}{2}$, which makes 99.99% of the values in $\mathbb{Z}_L$ valid, Π_{MW} achieves approximately a 2.5× improvement in efficiency compared to the baseline. Even for $B = 0.999999 \cdot \frac{L}{2}$, where only 0.0001% of the values on $\mathbb{Z}_L$ are invalid, Π_{MW} still provides nearly a 2× efficiency improvement over the baseline method. This implies that the efficiency of Π_{MW} can be significantly improved even there is a very small gap between B and $\frac{L}{2}$. For the case there is no prior knowledge on the range of x, the baseline method is used, which invokes only one l-bit comparison protocol.

Table 3: Communication and runtime for protocol $\Pi_{\mathsf{MW}}^{l,l'}$, where $l = 37$, $l' = 2$ and $L = 2^l$. The communication and runtime are accumulated for 2^{20} runs of the protocols.

B	Time (s)		Comm. (MB)
	LAN	WAN	
$0.5 \cdot \frac{L}{2}$	0.19	2.04	16.25
$0.8 \cdot \frac{L}{2}$	0.67	5.51	49.25
$0.9999 \cdot \frac{L}{2}$	4.94	28.19	262.25
$0.999999 \cdot \frac{L}{2}$	6.35	35.65	336.25
$1 \cdot \frac{L}{2}$ (baseline)	12.13	71.08	671.75

Table 4: Comparing the runtime and communication of division protocol in CrypTFlow2 [27] with ours. For $x \in \mathbb{Z}_L$, we consider constraint $B \in \{B_1, B_2, B_3, \frac{L}{2}\}$, where $B_1 = 0.5 \cdot \frac{L}{2}$, $B_2 = 0.99 \cdot \frac{L}{2}$ and $B_3 = 0.999999 \cdot \frac{L}{2}$. The communication and runtime are accumulated for 2^{18} runs of the protocols.

Method	B	Time (s)		Comm. (MB)
		LAN	WAN	
[27]	-	7.75	48.24	354.72
Ours	B_1	1.60	11.36	82.53
		4.84×	**4.24**×	**4.29**×
	B_2	2.12	15.28	113.09
		3.65×	**3.15**×	**3.13**×
	B_3	3.84	22.26	167.59
		2.01×	**2.16**×	**2.11**×
	$\frac{L}{2}$	5.46	32.99	252.72
		1.41×	**1.46**×	**1.40**×

6.2 Experiments on Real-world Functions

Table 5: Communication and runtime for our evaluating $\sin(x)$ protocol Π_{sin}. The bitwidth and precision are $l = 21$ and $f = 12$. For $x \in \mathbb{Z}_L$, the values of constraint B are set as $B_1 = 0.5 \cdot \frac{L}{2}$, $B_2 = 0.99 \cdot \frac{L}{2}$, $B_3 = 0.999999 \cdot \frac{L}{2}$ and $\frac{L}{2}$. Symbols "aULP" and "mULP" denote the average and maximum ULP error, respectively. The communication (in MB) and runtime are accumulated for 2^{18} runs of the protocols.

B	Time (s)		Comm.	aULP	mULP
	LAN	WAN			
B_1	4.73	43.44	381.59	0.500	1.280
B_2	5.52	46.64	407.84	0.505	1.295
B_3	6.77	55.06	461.09	0.504	1.304
$\frac{L}{2}$	7.07	55.76	474.46	0.504	1.308

6.2.1 Division protocol

For $x \in \mathbb{Z}_L$, we perform four experiments on our division protocol Π_{Div}, with the constraints $|x| < B$ for $B \in \{0.5 \cdot \frac{L}{2}, 0.9999 \cdot \frac{L}{2}, 0.999999 \cdot \frac{L}{2}, \frac{L}{2}\}$ and compare the performance with CrypTFlow2's division protocol [27]. The experimental results are presented in Table 4. When $B = 0.5 \cdot \frac{L}{2}$, Π_{Div} achieves the highest efficiency, demonstrating an improvement of 4.24× to 4.84× compared to CrypTFlow2's method. For $B = 0.999999 \cdot \frac{L}{2}$, our protocol outperforms CrypTFlow2 by 2.01× to 2.16×. Even for the case there is no extra constraint on $|x|$, Π_{Div} achieves an approximately 1.4× improvement. These experimental results demonstrate that the performance gains of our Π_{Div} are not only from the optimization of computing $\mathsf{MW}(x)$ protocol, but also from the novel design of the division protocol. Moreover, Π_{Div} produces the exact result $\lfloor \frac{x}{d} \rfloor$ without error.

6.2.2 $\sin(x)$ protocol

We conduct experiments on our trigonometric protocol Π_{sin}, with the experimental results presented in Table 5. The communication increases only slightly with the constraint B, as the primary overhead arises from the multiplication protocols. The experimental results also demonstrate the high accuracy of Π_{sin}, with maximum ULP error as approximately 1.3. Variations in the ULP error are attributed to the randomness of the input values in each experimental group.

6.2.3 Exponential protocol

The experimental results for our exponential protocol Π_{rExp} are presented in Table 6, with all precisions set as $f = 12$. When compared with SirNN [26], we adopted SirNN's settings by choosing $l = 16$, and $l = f + 4$. Under these settings, the input range of e^{-x} is $x \in [0, 8)$. Consequently, the DReLU protocol in Π_{rExp} can be removed. Our Π_{rExp} achieves a 2.84× to 5.53× improvement in efficiency compared to SirNN. For accuracy, although SirNN achieves high accuracy for evaluating e^{-x}, our method achieves even higher accuracy, with a maximum ULP error of 1.435 at a precision of $f = 12$, corresponding to an error of approximately $3 \cdot 10^{-4}$ in floating-point. The ULP error is computed by testing all the inputs in interval $[0, 8)$. When comparing with Bolt [25], we adjusted the bitwidth to $l = 37$. In this configuration, the DReLU protocol is invoked, incurring greater overhead in Π_{rExp} compared to the $l = 16$ case. Despite this, Π_{rExp} achieves a 2.84× to 3.17× efficiency improvement over Bolt. In terms of accuracy, we compute ULP error for e^{-x} with range $x \in (0, 1000]$. The maximum ULP error for our method is 1.435, significantly outperforming prior works.

6.2.4 $\mathsf{Softmax}$ protocol

For $\mathsf{Softmax}$ function, the input used in our experiments is a 128×768 matrix, where 128 $\mathsf{Softmax}$ functions are performed, each with input length as 768. This setup reflects the typical parameters of the $\mathsf{Softmax}$ function employed in

Table 6: Comparing the runtime and communication costs of our evaluating e^{-x} protocol with SirNN [26] and Bolt [25]. For SirNN, the bitwidth is set as $l = 16$, and in Bolt, $l = 37$. All the precision is set as $f = 12$. The communication and runtime are accumulated for 2^{18} runs of the protocols.

l	Method.	Time. (s)		Comm. (MB)	avg ULP	max ULP
		LAN	WAN			
16	SirNN [26]	10.30	50.49	501.12	1.183	2.640
	Ours	1.86	17.75	156.90	0.353	1.435
		5.53×	**2.84**×	**3.19**×		
37	Bolt [25]	15.09	98.22	944.28	0.010	8.681
	Ours	4.88	34.58	297.81	0.004	1.435
		3.09×	**2.84**×	**3.17**×		

BERT (Bidirectional Encoder Representations from Transformers) [7]. The experimental results are listed in Table 7. The performance improvements of our Softmax protocol stem from two key optimizations: our novel protocol for evaluating e^{-x}, and an optimized batch division protocol. These enhancements enable a 4.6× to 6.2× improvement compared to Iron [13]. When compared with Bolt [25], our protocol reduces communication costs to 42% of its original level, and achieves an approximately 2× improvement in runtime.

Table 7: Comparing the runtime and communication costs of our Softmax protocol with Bolt [25] and Iron [13]. The bitwidth and precision are set as $l = 37$ and $f = 12$. The input of Softmax protocol is a 128×768-dimensional matrix, where there are 128 vectors, each with dimension 768.

Method	Time (s)		Comm. (MB)
	LAN	WAN	
Iron [13]	29.67	174.11	1660.68
Bolt [25]	9.80	73.63	630.16
Ours	4.78	37.91	270.84

6.2.5 Evaluation with Silent-OT

Recently, silent-OT has been proposed as a new OT-extension technique [3]. Its distinguishing feature is that, after a single relatively small interaction (sending a short seed plus a small amount of data), both parties can then locally and "silently" expand the OT to a very large scale. Thus, although silent OT requires substantially more computation, it incurs very little communication overhead. There are many frameworks [15,21, 30] built on silent-OT that achieve substantial improvements in communication efficiency.

Note that our focus is on upper-layer algorithmic design and optimization, aiming to reduce either the OT count or the OT input length. Consequently, our protocol can also benefit from silent-OT, while still outperforming prior approaches under the same OT-extension technique. To evaluate the performance and improvement of our method in the silent-OT setting, we conduct experiments and compare the overhead of exponential protocol with BumbleBee [21], which use silent-OT and bOLE techniques, and the experimental results are shown in Table 8. This comparison show that our method outperform prior work by a factor 3.95× to 4.88× using the silent-OT. Note that in this comparison, both our method and BumbleBee use silent OT as the underlying primitive. Thus, this comparison is fair and all performance improvements stem from the new protocol design rather than from differences in the OT-extension technique.

Table 8: Comparing the runtime and communication costs of our evaluating e^{-x} protocol with Bumblebee [21] and ours using silent-OT. The bitwidth is set as $l = 32$, and precision is set as $f = 12$. The communication and runtime are accumulated for 2^{18} runs of the protocols.

Method	Time (s)		Comm (MB)
	LAN	WAN	
Bumblebee [21]	13.96	51.10	156.41
Ours	3.53	10.48	60.03
	3.95×	**4.88**×	**2.61**×

7 Conclusion

This work proposes a new method to design secure two-party protocols for a class of real-world functions with real-number inputs. We first generalize the computing $\mathsf{MW}(x)$ protocol to accommodate arbitrary constraint. Building on this foundation, we give an efficient method for computing signed real number from shares. Then we propose secure evaluation methods for a variety of real-world functions, including division, trigonometric, and exponential functions. Our approach enables these functions to be evaluated with both low overhead and high accuracy. This work introduces a novel perspective on protocol design, and we anticipate that the ideas presented here will inspire broader advancements in protocol development and optimization.

Acknowledgments

We thank the anonymous shepherd and reviewers for their valuable feedback. We also thank Chunzao Huang for the fruitful suggestions. This work was supported in part by National Natural Science Foundation of China (Grant No. 62301190, 62502464), Shenzhen Colleges and Universities Stable Support Program (Grant No. GXWD20231129135251001), Shenzhen Fundamental Research Program (Grant No. JCYJ20241202124023031).

Ethical Considerations

This work focuses on the optimization and design for the efficient secure two-party computation protocols. As underlying computation operations, our designs involve no ethical consideration. The ethical consideration may occur in applications. We show that improving the efficiency of secure multi-party computation can also give rise to certain ethical concerns.

- The techniques proposed in this paper can improve the efficiency of secure two-party computation while fully preserving privacy. This property, however, may be exploited by malicious or criminal organizations to evade oversight. For example, multiple malicious parties could combine their data to carry out illegal activities, but because all data are processed under encryption, effective regulation becomes difficult.
- Recently, some works have taken advantage of the inherent inefficiency of secure multi-party computation to design higher-level functionalities, such as the work in [8]. This work builds on the fact that the hash function is very efficient in plaintext but extremely slow in MPC. The techniques proposed in this paper improve the efficiency of secure multi-party computation and therefore, to some extent, may undermine this type of design.
- MPC protects only the privacy of the intermediate computation, not the privacy that may be revealed by the output itself. For example, in the classic millionaire's problem, the protocol leaks no information during execution, but once one party learns the comparison result, they can infer the possible range of the other party's value. Therefore, special attention is required when applying MPC.

Open Science

We fully endorse the principles of Open Science Policy. To promote transparency and reproducibility, we have integrated our research artifacts into an open-source repository on GitHub https://github.com/geralt-tian/GEO_2025 and https://github.com/geralt-tian/GEO_2025_VOLE, which is now ready for public access.

References

[1] Secure and correct inference (sci) library. https://github.com/mpc-msri/EzPC/tree/master/SCI.

[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*, pages 535–548, 2013.

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

[4] Gilles Brassard, Claude Crépeau, and Jean-Marc Robert. All-or-nothing disclosure of secrets. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 234–238. Springer, 1986.

[5] Ran Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptol.*, 13(1):143–202, 2000.

[6] Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni, and Michael Zohner. Pushing the communication barrier in secure computation using lookup tables. *IACR Cryptol. ePrint Arch.*, page 486, 2018.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[8] Stefan Dziembowski, Sebastian Faust, Tomasz Lizurej, and Marcin Mielniczuk. Secret sharing with snitching. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 840–853, 2024.

[9] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC

over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 823–852. Springer, 2020.

[10] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[11] Hao Guo, Zhaoqian Liu, Ximing Fu, and Zhusen Liu. SEAF: secure evaluation on activation functions with dynamic precision for secure two-party inference. In *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*, pages 3417–3435. USENIX Association, 2025.

[12] Hao Guo, Liqiang Peng, Haiyang Xue, Li Peng, Weiran Liu, Zhe Liu, and Lei Hu. Improved secure two-party computation from a geometric perspective. In *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*, pages 4957–4974. USENIX Association, 2025.

[13] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. Iron: Private inference on transformers. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[14] 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, Paper 2023/1147, 2023. https://eprint.iacr.org/2023/1147.

[15] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. Cheetah: Lean and fast secure two-party deep neural network inference. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 809–826. USENIX Association, 2022.

[16] 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.

[17] Andes Y. L. Kei and Sherman S. M. Chow. SHAFT: secure, handy, accurate and fast transformer inference. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025.

[18] Vladimir Kolesnikov and Ranjit Kumaresan. Improved OT extension for transferring short secrets. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 54–70. Springer, 2013.

[19] Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. In Yehuda Lindell, editor, *Tutorials on the Foundations of Cryptography*, pages 277–346. Springer International Publishing, 2017.

[20] 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.

[21] Wen-jie Lu, Zhicong Huang, Zhen Gu, Jingyu Li, Jian Liu, Cheng Hong, Kui Ren, Tao Wei, and Wenguang Chen. Bumblebee: Secure two-party inference framework for large transformers. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025.

[22] Wen-jie Lu, Zhicong Huang, Qizhi Zhang, Yuchen Wang, and Cheng Hong. Squirrel: A scalable secure two-party computation framework for training gradient boosting decision tree. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 6435–6451. USENIX Association, 2023.

[23] Payman Mohassel and Peter Rindal. Aby^3: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 35–52. ACM, 2018.

[24] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy, SP*

2017, San Jose, CA, USA, May 22-26, 2017, pages 19–38. IEEE Computer Society, 2017.

[25] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. BOLT: privacy-preserving, accurate and efficient inference for transformers. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, pages 4753–4771. IEEE, 2024.

[26] 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.

[27] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow2: Practical 2-party secure inference. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 325–342. ACM, 2020.

[28] Abhi Shelat and Chih-Hao Shen. Fast two-party secure computation with minimal assumptions. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 523–534. ACM, 2013.

[29] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *Proc. Priv. Enhancing Technol.*, 2021(1):188–208, 2021.

[30] Tianshi Xu, Wen-jie Lu, Jiangrui Yu, Yi Chen, Chenqi Lin, Runsheng Wang, and Meng Li. Breaking the layer barrier: Remodeling private transformer inference with hybrid CKKS and MPC. In Lujo Bauer and Giancarlo Pellegrino, editors, *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*, pages 2653–2672. USENIX Association, 2025.

[31] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 160–164. IEEE Computer Society, 1982.

[32] Chi Chih Yao. How to generate and exchange secrets. In *Symposium on Foundations of Computer Science*, 2008.

[33] Lijing Zhou, Ziyu Wang, Hongrui Cui, Qingrui Song, and Yu Yu. Bicoptor: Two-round secure three-party non-linear computation without preprocessing for privacy-preserving machine learning. *CoRR*, abs/2210.01988, 2022.

[34] Lijing Zhou, Bingsheng Zhang, Ziyu Wang, Tianpei Lu, Qingrui Song, Su Zhang, Hongrui Cui, and Yu Yu. On probabilistic truncation in privacy-preserving machine learning. In Toby Walsh, Julie Shah, and Zico Kolter, editors, *AAAI-25, Sponsored by the Association for the Advancement of Artificial Intelligence, February 25 - March 4, 2025, Philadelphia, PA, USA*, pages 22955–22964. AAAI Press, 2025.

A Proofs

A.1 Proof of Lemma 1

Proof. To prove this lemma, we need to prove that $x > y$ if and only if $\lfloor \frac{x}{A} \rfloor > \lfloor \frac{y}{A} \rfloor$. For necessity, $x > y$ if and only if $x \geq y + A$ as $x - y \in [A, L) \bigcup [-L, 0)$. Therefore $\frac{x}{A} \geq \frac{y}{A} + 1$, then we have $\lfloor \frac{x}{A} \rfloor > \lfloor \frac{y}{A} \rfloor$.

For sufficiency, we use the method of proof by contradiction. Suppose $x \leq y$, we have $\frac{x}{A} \leq \frac{y}{A}$ and $\lfloor \frac{x}{A} \rfloor \leq \lfloor \frac{y}{A} \rfloor$, which contradicts the condition $\lfloor \frac{x}{A} \rfloor > \lfloor \frac{y}{A} \rfloor$. Therefore, we have $x > y$. □

A.2 Proof of Lemma 2

Proof. As $x_0 + x_1 \neq L$, we have that $\mathsf{Wrap}(x_0, x_1, L) = \mathbf{1}\{x_0 + x_1 \geq L\} = \mathbf{1}\{x_0 + x_1 < L\} \oplus 1 = \mathsf{Comp}(x_1, L - x_0) \oplus 1$. As $A \neq 0$, we have $\mathsf{Comp}(x_1, L - x_0) \oplus 1 = \mathsf{Comp}(L - x_0, x_1)$. Let $a = x_1$ and $b = L - x_0$, we have $a - b = x_0 + x_1 - L \in [-L, 0) \bigcup [A, L)$. Then from Lemma 1 we can have that $\mathsf{Comp}(b, a) = \mathsf{Comp}(\lfloor \frac{b}{A} \rfloor, \lfloor \frac{a}{A} \rfloor)$. Therefore, $\mathsf{Wrap}(x_0, x_1, L) = \mathsf{Comp}(L - x_0, x_1) = \mathsf{Comp}(\lfloor \frac{L - x_0}{A} \rfloor, \lfloor \frac{x_1}{A} \rfloor)$. □

A.3 Proof of Theorem 1

Proof. For $|x| < B$, we have $x_0 + x_1 \in [0, B) \bigcup [L - B, L + B) \bigcup [2L - B, 2L)$. The theorem is proven by considering the following cases:

- For $x_0 + x_1 \in [0, B)$, $\mathsf{MW}(x) = 0$. In this case we have $x_0 \in [0, B)$, and thus $\Delta = 0$ and $x_0^* = x_0 - B + L$. Then $x_0^* + x_1 = x_0 + x_1 + L - B \in [L - B, L)$, which means $x_0^* + x_1 < 2L - 2B$ as $B < \frac{L}{2}$ and $M^* = 0$. Therefore, $\mathsf{MW}(x) = M^* + \Delta = 0$.
- For $x_0 + x_1 \in [L - B, L + B)$, $\mathsf{MW}(x) = 1$.
  - If $x_0 < B$, we have $\Delta = 0$ and $x_0^* = x_0 - B + L$. Then $x_0^* + x_1 = x_0 + x_1 + L - B \in [2L - 2B, 2L)$, and thus $M^* = 1$.

– If $x_0 > B$, we have $\Delta = 1$ and $x_0^* = x_0 - B$. Then $x_0^* + x_1 = x_0 + x_1 - B \in [L-2B, L)$, and thus $M^* = 0$.

Therefore, $\mathsf{MW}(x) = M^* + \Delta = 1$.

- For $x_0 + x_1 \in [2L-B, 2L)$, $\mathsf{MW}(x) = 2$. In this case $x_0 > L - B$, and thus $\Delta = 1$ as $B < \frac{L}{2}$. Then, $x_0^* + x_1 = x_0 + x_1 - B \in [2L-2B, 2L-B)$, which means $M^* = 1$. Therefore, $\mathsf{MW}(x) = M^* + \Delta = 2$.

In all cases, $\mathsf{MW}(x) = M^* + \Delta$, completing the proof. □

A.4 Proof of Lemma 3

Proof. We study the relationship between $\mathsf{MW}(z, 2^l)$ and $\mathsf{MW}(y, 2^{l+1})$ in the following cases.

- **Case 1:** $y_0 + y_1 \in [0, 2^{l-1})$ (point $P \in \mathcal{A}$). For $i \in \{0,1\}$, we have $y_i < 2^{l-1}$ and $z_i = y_i$. Therefore, $z_0 + z_1 = y_0 + y_1$ belongs to interval $[0, 2^{l-1})$, and $\mathsf{MW}(z, 2^l) = \mathsf{MW}(y, 2^{l+1}) = 0$.
- **Case 2:** $y_0 + y_1 \in [2^{l+1} - 2^{l-1}, 2^{l+1} + 2^{l-1})$. In this case, we consider the following sub-cases:
  - **Case (i):** $y_0 < 2^l$ and $y_1 < 2^l$ (point $P \in \mathcal{B}_0$). For $i \in \{0,1\}$, $z_i = y_i$. Therefore, $z_0 + z_1 = y_0 + y_1$, belongs to interval $[2^{l+1} - 2^{l-1}, 2^l)$. In this case, $b = 1$, $\mathsf{MW}(y, 2^{l+1}) = 1$, and $\mathsf{MW}(z, 2^l) = 2$. Thus, $\mathsf{MW}(z, 2^l) = \mathsf{MW}(y, 2^{l+1}) + b$.
  - **Case (ii):** $y_0 \geq 2^l$ and $y_1 \geq 2^l$ (point $P \in \mathcal{C}_0$). For $i \in \{0,1\}$, $z_i = y_i - 2^l$. Therefore, $z_0 + z_1 = y_0 + y_1 - 2^{l+1}$, belongs to interval $[0, 2^{l-1})$. In this case, $a = 1$, $\mathsf{MW}(y, 2^{l+1}) = 1$, and $\mathsf{MW}(z, 2^l) = 0$. Thus, $\mathsf{MW}(z, 2^l) = \mathsf{MW}(y, 2^{l+1}) - a$.
  - **Case (iii):** $y_0 \geq 2^l$ and $y_1 < 2^l$, or $y_0 < 2^l$ and $y_1 \geq 2^l$. In this case, $z_0 + z_1 = y_0 + y_1 - 2^l$, belongs to interval $2^l - 2^{l-1}, 2^l + 2^{l-1}$. Therefore, $\mathsf{MW}(z, 2^l) = \mathsf{MW}(y, 2^{l+1}) = 1$.
- **Case 3:** $y_0 + y_1 \in [2^{l+2} - 2^{l-1}, 2^{l+2})$ (point $P \in \mathcal{D}$). For $i \in \{0,1\}$, we have $y_i \geq 2^{l+1} - 2^{l-1}$, and $z_i = y_i \mod 2^l = y_i - 2^l$. Therefore, $z_0 + z_1 = y_0 + y_1 - 2^{l+1}$ belongs to interval $2^{l+1} - 2^{l-1}, 2^{l+1}$, which means $\mathsf{MW}(z, 2^l) = \mathsf{MW}(y, 2^{l+1}) = 2$.

Note that both a and b are zeros except for case (i) and case (ii). Therefore, we have $\mathsf{MW}(z, 2^l) = \mathsf{MW}(y, 2^{l+1}) - a + b$. □

A.5 Proof of Theorem 2

Proof. Let $\hat{y} = \hat{y}_0 + \hat{y}_1$. The range of $\hat{y}_0 + \hat{y}_1$ is $\hat{y}_0 + \hat{y}_1 \in [0, 2^{l-1}) \bigcup [2^l - 2^{l-1}, 2^l + 2^{l-1}) \bigcup [2^{l+1} - 2^{l-1}, 2^{l+1})$, corresponding to $\mathsf{MW}(\hat{y}, 2^{l+1})$ takes value of 0, 1 and 2, respectively. Moreover, $\hat{y}_0 + \hat{y}_1 \in [0, 2^{l-1})$ if and only if $b = 1$, and $\hat{y}_0 + \hat{y}_1 \in [2^{l+1} - 2^{l-1}, 2^{l+1})$ if and only if $a = 1$. Therefore, $a - b$ can be written as:

$$a - b = \begin{cases} 1, & \mathsf{MW}(\hat{y}, 2^{l+1}) = 0 \\ 0, & \mathsf{MW}(\hat{y}, 2^{l+1}) = 1 \\ -1, & \mathsf{MW}(\hat{y}, 2^{l+1}) = 2 \end{cases},$$

which can be summarized as $a - b = 1 - \mathsf{MW}(\hat{y}, 2^{l+1})$. We use the idea in Theorem 1 to compute $\mathsf{MW}(\hat{y})$, with the constraint $|y| < 2^l$ for $y \in \mathbb{Z}_{2^{l+1}}$. Defining $\hat{M}^* = \mathbf{1}\{\hat{y}_0^* \geq 2^l\} \wedge \mathbf{1}\{\hat{y}_1 \geq 2^l\}$ and $\delta = \mathbf{1}\{\hat{y}_0 \geq 2^{l-1}\}$, we have $\hat{M}^* = \mathbf{1}\{\hat{y}_0^* + y_1 \geq 2^{l+1}\} = \mathbf{1}\{\hat{y}_0^* \geq 2^l\} \wedge \mathbf{1}\{\hat{y}_1 \geq 2^l\}$ and $\mathsf{MW}(\hat{y}) = \hat{M}^* + \delta$. Therefore, $a - b = 1 - \mathsf{MW}(\hat{y}) = 1 - \delta - \hat{M}^*$. □

B Evaluating $\sin(x)$ protocol

Our evaluating $\sin(x)$ protocol $\Pi_{\sin}$ is shown in Algorithm 6. In the multiplication protocol used in lines 4–7, the inputs are held by P_0 and P_1 locally. Therefore, the method in subsection 3.1.4 is used. Moreover, we can use the known MSB method to improve the efficiently. For instance, when computing the product $s_0 \cdot c_1$, we first set $\hat{s}_0 = s_0 + 2^f$ and $\hat{c}_1 = c_1 + 2^f$, ensuring both values are non-negative. Then, the product $\hat{s}_0 \cdot \hat{c}_1$ can be computed using a single invocation of $\mathcal{F}_{\mathsf{CrossTerm}}$. The original product $s_0 \cdot c_1$ can be recovered via $s_0 \cdot c_1 = \hat{s}_0 \cdot \hat{c}_1 - 2^f \cdot c_1 - 2^f \cdot s_0 - 2^{2f}$, where the last three terms can be computed locally. When computing $(sc + cs) \cdot C_j$ and $(cc - ss) \cdot S_j$ for $j \in \{0,1,2\}$, the signed extension protocols $\mathcal{F}_{\mathsf{SExt}}$ are invoked since C_j and S_j can be seen as public constants (see Section 3.1.4 for details). Only two $\mathcal{F}_{\mathsf{SExt}}$ are required in the loop, one for $sc + cs$ and one for $cc - ss$, as these values are unchanged. Moreover, as the ranges of $(sc + cs)$ and $(cc - ss)$ are small, $\mathcal{F}_{\mathsf{SExt}}$ can be implemented using the method in [12], resulting in very small overhead. Finally, after retrieving the value of $\sin(x)$ from the lookup table, an additional $\mathcal{F}_{\mathsf{SExt}}$ is invoked to extend the bitwidth to l', which can also be implemented using the method from [12].

Correctness and security. The correctness of $\Pi_{\sin}$ is ensured by Equation 12. The security comes from the security of protocols for $\mathcal{F}_{\mathsf{Mul}}$, $\mathcal{F}_{\mathsf{LUT}}$, $\mathcal{F}_{\mathsf{MW}}$, and $\mathcal{F}_{\mathsf{SExt}}$.

Complexity. $\Pi_{\sin}^{l,f,l',f'}$ invoke protocols for four $\mathcal{F}_{\mathsf{Mul}}^{l_t,l_t}$, two $\mathcal{F}_{\mathsf{Mul}}^{2l_t,l_T}$, one $\mathcal{F}_{\mathsf{MW}}^{l,2}$, one $\mathcal{F}_{\mathsf{LUT}}$, and one $\mathcal{F}_{\mathsf{SExt}}^{3+f',l'}$. The four $\mathcal{F}_{\mathsf{Mul}}^{l_t,l_t}$ can be realized by four $\mathcal{F}_{\mathsf{CrossTerm}}^{l_t,l_t}$, with communication $4 \cdot (l_t \cdot (\lambda + \frac{l_t}{2} + \frac{1}{2}) + l_t^2)$. The two $\mathcal{F}_{\mathsf{Mul}}^{2l_t,l_T}$ can be realized by two $\mathcal{F}_{\mathsf{SExt}}^{2l_t,2l_t+l_T}$, with communication $2(\lambda + l_T)$. The communication for $\mathcal{F}_{\mathsf{MW}}$ is denoted as C_{MW}. Communication of protocols for $\mathcal{F}_{\mathsf{LUT}}$ and the last one $\mathcal{F}_{\mathsf{SExt}}^{5+f',l'}$ is $2\lambda + 4(2l_t + l_T)$ and $\lambda + l' - f - 3$. Therefore, the total communication for $\Pi_{\sin}^{l,f,l',f'}$ is approximately $\lambda(4l_t + 5) + 6l_t^2 + 10l_t + 6l_T + C_{\mathsf{MW}}$.

Algorithm 6: Computing $\sin(x)$, $\Pi_{\mathsf{sin}}^{l,f,l',f'}$:

Input: P_0 and P_1 hold $[\![x]\!]^l$ with precision f.

Output: P_0 and P_1 output $[\![z]\!]^{l'}$ with precision f', such that $\mathsf{Real}(z) = \sin(\mathsf{Real}(x))$.

Let $f_t = 14$, $l_t = f_t + 2$, $f_T = 30$ and $l_T = f_T + 2$.

For $i \in \{0,1\}$, P_i computes $\sin(\frac{x_i}{2^f})$ and $\cos(\frac{x_i}{2^f})$, and encode them as $s_i = \mathsf{Fix}(\sin(\frac{x_i}{2^f}), l_t, f_t)$ and $c_i = \mathsf{Fix}(\cos(\frac{x_i}{2^f}), l_t, f_t)$.

P_0 and P_1 invoke the following functionalities:

- $\mathcal{F}_{\mathsf{Mul}}^{l_t,l_t}(s_0, c_1)$ to learn $[\![sc]\!]^{2l_t}$.
- $\mathcal{F}_{\mathsf{Mul}}^{l_t,l_t}(c_0, s_1)$ to learn $[\![cs]\!]^{2l_t}$.
- $\mathcal{F}_{\mathsf{Mul}}^{l_t,l_t}(c_0, c_1)$ to learn $[\![cc]\!]^{2l_t}$.
- $\mathcal{F}_{\mathsf{Mul}}^{l_t,l_t}(s_0, s_1)$ to learn $[\![ss]\!]^{2l_t}$.

for $j \in \{0,1,2\}$ **do**

- P_0 and P_1 compute $C_j = \cos(-j \cdot \frac{L}{2^f})$ and $S_j = \sin(-j \cdot \frac{L}{2^f})$.
- P_0 and P_1 invoke $\mathcal{F}_{\mathsf{Mul}}^{2l_t,l_T}([\![sc]\!]^{2l_t} + [\![cs]\!]^{2l_t}, C_j)$ to learn $[\![temp_0]\!]^{2l_t+l_T}$.
- P_0 and P_1 invoke $\mathcal{F}_{\mathsf{Mul}}^{2l_t,l_T}([\![cc]\!]^{2l_t} - [\![ss]\!]^{2l_t}, S_j)$ to learn $[\![temp_1]\!]^{2l_t+l_T}$.
- P_0 and P_1 set $[\![T]\!]^{2l_t+l_T}[j] = ([\![temp_0]\!]^{2l_t+l_T} + [\![temp_1]\!]^{2l_t+l_T})$.
- P_0 and P_1 set $[\![T]\!]^{5+f'}[j] = [\![T]\!]^{2l_t+l_T}[j] \gg (2f_t + f_T - f') \bmod 2^{5+f'}$.

end

P_0 and P_1 invoke $\mathcal{F}_{\mathsf{MW}}^{l,2}([\![x]\!]^l)$ to learn $[\![\mathsf{MW}]\!]^2$.

P_0 and P_1 invoke $\mathcal{F}_{\mathsf{LUT}}([\![T]\!]^{5+f'}, [\![\mathsf{MW}]\!]^2)$ to learn $[\![sin]\!]^{5+f'}$.

P_0 and P_1 invoke $\mathcal{F}_{\mathsf{SExt}}^{5+f',l'}([\![sin]\!]_i^{5+f'})$ to learn $[\![sin]\!]_i^{l'}$.

P_0 and P_1 output $[\![sin]\!]^{l'}$.

C Ideal Functionalities

For our computing $\mathsf{MW}(x)$ protocol, the ideal functionality $\mathcal{F}_{\mathsf{MW}}$ is defined as follows.

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

Constraint: For $x \in \mathbb{Z}_L$, there is a known upper bound $B \leq \frac{L}{2}$, imposing $|x| < B$. If there is no prior knowledge about the range of x, B is set as $\frac{L}{2}$.

- $\mathcal{F}_{\mathsf{MW}}$ receives x_0 from P_0 and x_1 from P_1.
- $\mathcal{F}_{\mathsf{MW}}$ computes $mw = \mathsf{MW}(x_0, x_1, L)$ according to Equation 1.
- $\mathcal{F}_{\mathsf{MW}}$ shares mw to P_0 and P_1, respectively.

Based on $\mathsf{MW}(x)$ and $\mathsf{Real}(x)$, we can evaluate functions $\mathsf{func}(\cdot)$ with the property listed in Equation 3. The ideal functionality for evaluate $\mathcal{F}_{\mathsf{func}}$ is listed below, and the functionalities for evaluate integer division, trigonometric and exponential functions are instances of $\mathsf{func}(\cdot)$.

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

Constraint: For $x \in \mathbb{Z}_L$, there is a known upper bound $B \leq \frac{L}{2}$, imposing $|x| < B$. If there is no prior knowledge about the range of x, B is set as $\frac{L}{2}$.

- $\mathcal{F}_{\mathsf{func}}$ receives x_0 from P_0 and x_1 from P_1.
- $\mathcal{F}_{\mathsf{func}}$ computes $mw = \mathsf{MW}(x_0, x_1, L)$ according to Equation 1 and $res = \sum_{i=0}^{k-1} f_i(\frac{x_0}{2^f}) \cdot g_i(\frac{x_1}{2^f}) \cdot h_i(\frac{-\mathsf{MW}(x) \cdot L}{2^f})$.
- $\mathcal{F}_{\mathsf{func}}$ shares res to P_0 and P_1, respectively.