2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179449

Bicoptor: Two-round Secure Three-party Non-linear Computation without Preprocessing for Privacy-preserving Machine Learning

Lijing Zhou*, Ziyu Wang‡*, Hongrui Cui†, Qingrui Song* and Yu Yu†
*Huawei Technology, Shanghai, China
{zhoulijing,wangziyu13,songqingrui1}@huawei.com
†Shanghai Jiao Tong University, Shanghai, China
{rickfreeman,yyuu}@sjtu.edu.cn

***Abstract*—The overhead of non-linear functions dominates the performance of the secure multiparty computation (MPC) based privacy-preserving machine learning (PPML). This work introduces a family of novel secure three-party computation (3PC) protocols, Bicoptor, which improve the efficiency of evaluating non-linear functions. The basis of Bicoptor is a new sign determination protocol, which relies on a clever use of the truncation protocol proposed in SecureML (S&P 2017). Our 3PC sign determination protocol only requires two communication rounds, and does not involve any preprocessing. Such sign determination protocol is well-suited for computing non-linear functions in PPML, e.g. the activation function ReLU, Maxpool, and their variants. We develop suitable protocols for these non-linear functions, which form a family of GPU-friendly protocols, Bicoptor. All Bicoptor protocols only require two communication rounds without preprocessing. We evaluate Bicoptor under a 3-party LAN network over a public cloud, and achieve more than 370,000 DReLU/ReLU or 41,000 Maxpool (find the maximum value of nine inputs) operations per second. Under the same settings and environment, our ReLU protocol has a one or even two orders of magnitude improvement to the state-of-the-art works, Falcon (PETS 2021) or Edabits (CRYPTO 2020), respectively without batch processing.**

1. Introduction

Secure multiparty computation (MPC) [1], [2], [3], [4], [5] is a fundamental cryptographic primitive that allows multiple parties to jointly evaluate any efficiently computable functions while preserving the input secrecy. An area that raises particular privacy concerns is machine learning (ML) where the predictive model is typically acquired by aggregating and analyzing sensitive data from numerous institutions. Moreover, performing inference operations on the model may also impose privacy concerns since mobile or IoT devices typically outsource sensitive data to cloud ML services.

Recently, MPC-based privacy-preserving machine learning (PPML), which strives to combine the utility of ML and the privacy-preserving guarantee of MPC, has received phenomenal attention from the research community. The key issue with this approach is the performance, since MPC would incur extra overhead on top of the considerably heavy ML operations. Various works aim to improve the performance of MPC PPML with different settings. Protocols such as Delphi [6], GAZZLE [7], CryptFlow2 [8], ABY2.0 [9], and Chameleon [10] lie in two parties realm. SecureNN [11], Falcon [12], CryptGPU [13], ABY3 [14], ASTRA [15], BLAZE [16], and CryptFlow [17] involve three parties. Fantastic [18], SWIFT [19], FLASH [20], and Trident [21] are executed among four parties.

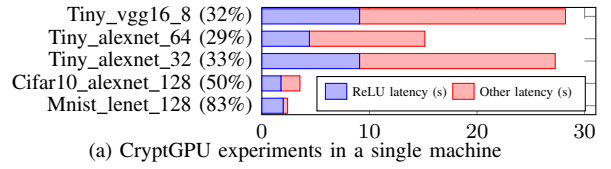
Among published works, CryptGPU [13] represents the state-of-the-art. We deploy the CryptGPU implementation [22] to run a sample PPML inference on both a single machine and a 3-party LAN network environment. The resulting runtimes are shown in Fig. 1. We notice that under different network environments, the computation of Rectified Linear Unit (ReLU) takes a large portion of the overall runtime, ranging from around one-third (local) to three-quarters (LAN). This would be exacerbated in commercial deployment settings where WAN network offers an even worse network environment.

The computation of non-linear functions (including ReLU) in CryptGPU is realized by the ABY3-based protocol [14] which is heavy in terms of communication overhead. In particular, the protocol for ReLU takes $3 + \log_2 \ell$ communication rounds and 45ℓ bits of bandwidth, for a input $x \in \mathbb{Z}_q$ and $\ell := \log_2 q$. The ReLU function can be decomposed to Derivative ReLU (DReLU), i.e. a comparison between the input and zero (or determine the sign of the input), and a multiplication. We focus on DReLU since multiplication is a common task in MPC which already has highly optimized solutions. Currently, MPC-based comparison (CMP) protocols could be categorized into four types.

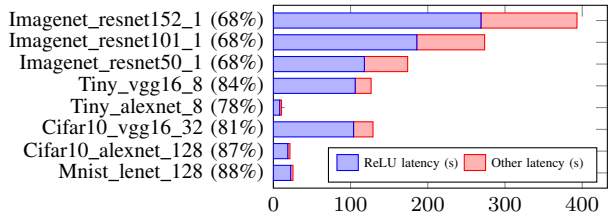
- **A2B-CMP-B2A**: First switch input sharing from arithmetic form to binary form (A2B), then perform the bit-wise comparison to obtain the binary shares of the comparison result, and finally switch back to the arithmetic form (B2A) [13], [14], [19]. Most notably, the ReLU protocol in CryptGPU utilizes this comparison method.

‡ *Ziyu Wang is the corresponding author*

DOI 10.1109/SP46215.2023.00074



(a) CryptGPU experiments in a single machine



(b) CryptGPU experiments with three machines over LAN

Figure 1: The CryptGPU experiments are named by the dataset_modelname_batchsize, e.g., Tiny_vgg16_8 corresponds to the vgg16 model trained from the Tiny dataset, and the inference runs with a batch size of 8. The percentage after the name reflect the ratio of the ReLU latency to the total latency, e.g., ReLU spends 32% of latency among the Tiny_vgg16_8 experiment in the local environment.

- **GC-based-CMP**: Directly apply the generic GC-based comparison protocol [7], [9], [15], [21], [23], [24].
- **Random-masking-CMP**: Open the secret input masked by a random r, i.e., $e := x + r$. Then the comparison of x against any constant c can be enabled by comparing r against $e - c$. The comparison can be aided by additional auxiliary preprocessing information generated alongside r [11], [12].
- **MSB-decomposition-CMP**: Decompose the input shares into the binary form and perform comparison [25].

Most of the works in the A2B-CMP-B2A, Random-masking-CMP, and MSB-decomposition-CMP categories require $\mathcal{O}(\log \ell)$ communication rounds (ℓ is the bit length of ring/field size). Despite that GC-based-CMP has a constant number of communication rounds, its bandwidth cost is usually the highest. Intuitively, communication is the bottleneck for all four comparison methods.

The performance of computing the Maxpool function is another bottleneck in PPML. In machine learning, it is usually required to determine the maximum element in a 2×2 or 3×3 matrix. Maxpool function involves several comparisons and dot products for element selections. Existing works in the literature mostly fall into two categories.

- **Repeated comparison**: The greater of the first two elements are compared with the third element and so on [11], [12].
- **Binary search**: Perform comparison in a binary tree manner, where the inputs are the leaf nodes and the maximum is at the root [25].

Like ReLU, protocols for Maxpool usually require dozens of communication rounds and Maxpool is another performance bottleneck for PPML.

We propose a novel and more efficient sign determination protocol, which implies a comparison protocol, to accelerate the overhead of non-linear functions like ReLU and Maxpool, in order to further optimize the performance of PPML. Our approach departs from existing methods, and improves the communication overhead compared to previous works.

Our high-level idea for the comparison protocol is as follows. For an input $x \in [0, 2^{\ell_x}) \bigcup (q - 2^{\ell_x}, q)$, we define a variable $\xi = \xi(x) := x$ if $x \in [0, 2^{\ell}_x)$ or $\xi = \xi(x) := q - x$ if $x \in (q - 2^{\ell_x}, q)$. ξ can be recognized as the "absolute value" of x. We define the bit position of the most significant non-zero bit of ξ as $\lambda - 1$. We further define λ as the effective bit length of ξ. [1] The truncation protocol $\mathrm{TRC}(x, k)$ represents truncating k-bits from x, which is originally proposed SecureML [24]. Our idea is to study the outcome of $\mathrm{TRC}(x, \lambda-1)$ or $\mathrm{TRC}(x, \lambda)$. If the outcome is 1 or $q-1$, then the input is positive or negative, respectively. Unfortunately, the truncation protocol proposed in SecureML [24] may introduce a one-bit error, which poses a problem to our protocol. By carefully analyzing the behavior of the errors, we explicitly identify the conditions under which the errors occur (Lemma 1 and 2), and prove the inevitable existence of 1 or $q - 1$ even if errors occur.

Regarding the fact that the input value is unknown in an MPC context, and hence the λ is unknown to all participants, we rely on performing repeated times of probabilistic truncations to compute the array $\{\mathrm{TRC}(x, 1), \cdots, \mathrm{TRC}(x, \lambda - 1), \mathrm{TRC}(x, \lambda), \cdots, \mathrm{TRC}(x, \ell_x)\}$, where ℓ_x is the precision of inputs. By using this method, we manage to construct a two-round 3-party sign determination protocol with better performance without relying on preprocessing. [2]

Based on the sign determination protocol, we further develop suitable 3PC protocols without preprocessing for common non-linear functions in PPML, e.g. ReLU, Maxpool, and their variants. It is worth mentioning that, the number of communication rounds of all our protocols is constant, i.e., 2, whereas the Maxpool protocols in Falcon [12], SecureNN [11], and CryptFlow [17] require 104, 72, and 72 communication rounds in a typical setting ($n = 9, \ell = 40$) respectively.

1.1. Related works

In MPC-based PPML, the overhead of evaluating the non-linear functions, e.g., ReLU and Maxpool, dominates the total overhead. Existing protocols, such as ABY3 [14], Edabits [25], CryptGPU [13], Fantastic [18], SWIFT [19], BLAZE [16], FLASH [20], Trident [21], and Falcon [12] mostly resort to preprocessing to enhance the online performance. In particular, after running an input-independent preprocessing phase which typically utilizes heavy cryptographic machinery, the parties are able to accomplish the

1. For example, for $\xi = x = 23 = 0b00010111$, $\ell_x = 8$, $\lambda = 5$, and $\xi_{\lambda-1} = \xi_4$.

2. In general, MPC offline phase includes both preprocessing and distributing shared randomness. The overhead of distributing shared randomness (usually one time) is much cheaper than that of preprocessing. It is worth distinguishing between these two ideas for the rest of the paper. Most previous MPC-based PPML works [12], [13], [14], [16], [18], [19], [20], [21], [25] require preprocessing which is heavily computed.

TABLE 1: The comparison between the communication overhead of our ReLu/Maxpool protocol and that of other related works. (ss/gc: secret sharing/garble circuit. Comm.: the dominant one-pass communication cost is counted in bits. ℓ: the bit length of $\mathbb{Z}_q = [0, q-1]$ ($\ell := \log_2 q$), e.g., $\ell = 40$. p: a prime field modulus, e.g., $p = 67, \lceil \log_2 p \rceil = 7$. ℓ_x: the precision of the input, e.g., $\ell_x = 16$. κ: the computational security parameter, e.g., $\kappa = 128$. s: the statistical security parameter, e.g., $s = 40$. n: the number of inputs. 2R: 2-round. BT: binary tree.)

ReLU

Protocol	Prep.	Round	Comm.(bit)
ABY_2PC [23]	Yes	5	$(2\kappa+20)\ell$
ABY2.0_2PC [9]	Yes	4	$(\kappa+3)\ell$
EMP_2PC [26]	No	2	$18\kappa\ell - 6\kappa$
CryptFlow2_2PC [8]	Yes	$4+\log\ell$	$32(\ell+1)+31\ell$
Fantastic_3PC [18]	Yes	$3+\log\ell$	$114\ell+6s+1$
BLAZE_ss_3PC [16]	Yes	$3+\log\ell$	16ℓ
BLAZE_gc_3PC [16]	Yes	4	$(\kappa+7)\ell$
SWIFT_3PC [19]	Yes	$3+\log\ell$	16ℓ
Falcon_3PC [12]	Yes	$5+\log\ell$	32ℓ
ABY3_3PC [14]	Yes	$3+\log\ell$	45ℓ
CryptFlow_3PC [17]	No	10	$(6\log p+19)\ell$
SecureNN_3PC [11]	No	10	$(8\log p+24)\ell$
CryptGPU_3PC [13]	Yes	$3+\log\ell$	45ℓ
Edabits_3PC [25]	Yes	$5+\log\ell$	80ℓ
Ours_3PC	**No**	**2**	$\boldsymbol{(\ell_x+2)\ell}$
Fantastic_4PC [18]	No	$1+\log\ell$	$44\ell+1$
SWIFT_4PC [19]	Yes	$1+\log\ell$	10ℓ
FLASH_4PC [20]	Yes	$2+\log\ell$	28ℓ
Trident_4PC [21]	Yes	4	$8\ell+4$

Maxpool

Protocol	Prep.	Round	Comm.(bit)
SWIFT_3PC [19]	Yes	$\log n(3+\log\ell)$	$(n-1)\cdot 16\ell$
Falcon_3PC [12]	Yes	$(n-1)(7+\log\ell)$	$(n-1)\cdot 32\ell$
CryptFlow_3PC [17]	No	$9(n-1)$	$(n-1)(6\log p+19)\ell$
SecureNN_3PC [11]	No	$10(n-1)$	$(n-1)(8\log p+24)\ell$
Ours_2R_3PC	**No**	**2**	$\boldsymbol{\frac{n(n-1)}{2}(\ell_x+2)\ell}$
Ours_BT_3PC	**No**	$\boldsymbol{\log n}$	$\boldsymbol{n(\ell_x+2)\ell}$
SWIFT_4PC [19]	Yes	$\log n(1+\log\ell)$	$(n-1)\cdot 16\ell$

PPML task relatively faster in the online phase once the inputs are ready. Notice that the total overhead (preprocessing and online) remains unchanged, i.e., with an improved performance for the online phase, the overhead of the preprocessing phase is usually heavy. For instance, Escudero et al. [25] propose a comparison method where the online comparison performance could be improved by preprocessed material called "Edabits". The generation of Edabits relies on homomorphic encryption or oblivious transfer which incurs significant performance overhead. Our work aims to optimize the overall performance of different non-linear functions used in PPML.

As discussed above, the overhead of evaluating the ReLU and Maxpool functions accounts for a large portion of the total overhead of an inference in PPML, and the comparison operation is the core of ReLU and Maxpool. We thus review different secure comparison methods. Let the input $x \in \mathbb{Z}_q$ where $\log_2 q = \ell$, the overheads of the four mainstream comparison protocol types are as follows. Note that "the comparison between two secrets", "the sign determination", and "the comparison between a secret and a constant" are equivalent in some way.

- **A2B-CMP-B2A**: ABY3 [14], SWIFT [19], and CryptGPU [13] first transform the secret input from arithmetic-form to Boolean form (A2B), then perform the bit-wise comparison, followed by a reversed transformation (B2A). This method usually takes $\mathcal{O}(\log\ell)$ rounds and communicates $\mathcal{O}(1)$ or $\mathcal{O}(\ell)$ bits in $\mathbb{Z}_q$. For example, the ReLU protocol in SWIFT [19] takes $3+\log\ell$ rounds and 10ℓ bits,
- **GC-based-CMP**: GAZZLE [7], ASTRA [15], ABY2.0 [9], ABY [23], SecureML [24], and Trident [21] apply the classical generic Yao garble-circuit (GC) method [1] to the secure comparison problem. Despite its optimized round overhead, the communication amount is usually significant. For instance, the communication bandwidth using EMP [26] is $18\kappa\ell - 6\kappa = 61,440$ bits under a typical setting of $\kappa = 128, \ell = 40$.
- **Random-masking-CMP**: In Falcon [12] and SecureNN [11] the shares of an input x is masked by a random r. The masked input $e := x + r$ is then made public, and the result of comparing r with $q/2 - e$ thus reflects the relation between x and $q/2$, which implies the sign of x. The overhead of this method is $\mathcal{O}(1)$ or $\mathcal{O}(\log\ell)$ rounds and $\mathcal{O}(\ell)$ bits, e.g, Falcon [12]'s ReLU spends $5+\log\ell$ rounds and 32ℓ bits, and SecureNN [11]'s ReLU requires 10 rounds and $8\log p + 24\ell$ bits, where p is a small field modulus (e.g., $p = 67$).
- **MSB-decomposition-CMP**: In Edabits [25], the shares of x, e.g., $[x]_0$ and $[x]_1$, is represented by $[x]_0 := [x]_0'' \cdot \lceil \frac{q}{2} \rceil + [x]_0'$ and $[x]_1 := [x]_1'' \cdot \lceil \frac{q}{2} \rceil + [x]_1'$. Then, checking whether $[x]_0' + [x]_1' \geq \lceil \frac{q}{2} \rceil$ and computing $[x]_0'' \oplus [x]_1''$ offers the shares of two intermediate values $[\mathsf{temp1}]$ and $[\mathsf{temp2}]$, respectively. The sign of x is obtained by $1 - [\mathsf{temp1} \oplus \mathsf{temp2}]$. The Edabits [25] communication overheads are $5+\log\ell$ rounds and 80ℓ bits.

All protocols except the GC-based ones take more than two rounds of communication. Nevertheless, the GC-based method takes considerably more communication bandwidth compared to the secret-sharing-based ones. Thus our work enjoys a significant round complexity advantage compared to prior works.

For the maxpool layer in PPML, we investigate the performance of protocols for the functionality of finding the maximum element.

- **Repeated comparison**: The maximum protocols in CryptFlow [17], Falcon [12], and SecureNN [11] get the output by sequentially comparing the output from the previous comparison to the next input element. Intuitively, the sequential operations take $\mathcal{O}(n)$ rounds. In particular, Falcon [12]'s Maxpool requires $(n-1)\cdot$

$(7 + \log \ell)$ rounds and $(n-1) \cdot 32\ell$ bits.

- **Binary search**: SWIFT [19] follows this method, in which the comparison is recursively applied to every different pair of inputs, until the output is narrowed down to the maximum. The protocol takes $\log n \cdot (3 + \log \ell)$ rounds and $(n-1) \cdot 16\ell$ bits.

Again, our Maxpool protocols outperform other protocols using these two methods in terms of round complexity. We compare the number of communication rounds and the cost of the dominant one-way bandwidth in our ReLU and Maxpool protocols against those in the literature in Tab. 1.

1.2. Our contributions

We summarize the contributions of this work as follows.

- We define a variable $\xi = \xi(x) := x$ if an input $x \in [0, 2^{\ell}_{x})$ or $\xi = \xi(x) := q - x$ if $x \in (q - 2^{\ell_x}, q)$, and λ refers to the effective bit length of ξ. We use $\mathrm{TRC}(x, \lambda - 1)$ or $\mathrm{TRC}(x, \lambda)$ to represent truncating λ or $\lambda - 1$ bits from x. We prove the inevitable existence of 1 or $q-1$ for a positive or a negative input even if errors occur (Lemma 3). We further show the maximum numbers of truncations required to compute $\mathrm{TRC}(x, \lambda - 1)$ or $\mathrm{TRC}(x, \lambda)$ in Lemma 4. Based on Lemma 3 and Lemma 4, we design a novel two-round 3PC DReLU protocol without preprocessing.
- After applying some optimization techniques, we further extend the DReLU protocol to other non-linear functions in PPML, including the Equality, ABS, ReLU, Dynamic ReLU (Leaky ReLU, PReLU, RReLU), ReLU6, Piecewise Linear Unit (PLU), MAX, MIN, SORT, and median (MED) functions. By carefully merging the multiplication operation(s) into the DReLU operation, we manage to achieve two rounds of communication for all the aforementioned protocols. In comparison, the corresponding ReLU or MAX protocols in prior works usually have dozens of rounds.
- We implement all the protocols and evaluate their performance under LAN settings (three VMs in the same cloud region). Without batching, our DReLU/ReLU protocols achieve a one or two orders of magnitude improvement compared with the state-of-the-art work Falcon [12] or Edabits [25], respectively. [3] When the batch size is 100,000, we achieve 390,000 DReLU or 370,000 ReLU operations per second.

Most existing MPC protocols to evaluate non-linear functions rely on sequentially dependent computation, e.g., A2B switching or GC, and hence are not quite GPU-friendly. In comparison, all of our Bicoptor protocols (e.g., ReLU, CMP, and Maxpool) are suitable for GPU implementation since most of their computation steps are parallelizable.

For the rest of the paper, Sect. 2 introduces the notations, system settings, related backgrounds, and a key building block, i.e., the truncation protocol. Sect. 3 intuitively presents the novel DReLU protocol. Sect. 4 extends DReLU to other non-linear functions in PPML. Sect. 5 analyses the concrete overhead of our protocols, and exhibits the performance of our implementations compared with Edabits [25] and Falcon [12]. Finally, Sect. 6 concludes this paper. In Appendix, App. A introduces the fixed-point computation to handle decimal arithmetic. App. B presents the non-linear functions used in this paper. App. C presents the proofs of all lemmas used in this paper. App. D proposes a concrete example for our DReLU protocol.

3. We take the honest-majority and passive security settings. The ring size and the precision of inputs are set to 64 bits and 13 bits, respectively.

TABLE 2: Notation table.

Notation	Description
$:=$	defined as
i, j, k	indexes
x, y or $\{x_i\}$	a single input x or y, or an input array $\{x_i\}$
ξ	if an input $x \in [0, 2^{\ell_x})$ then $\xi = \xi(x) := x$; if an input $x \in (q - 2^{\ell_x}, q)$ then $\xi = \xi(x) := q - x$; ξ can be recognized as the "absolute value" of x; the binary form of ξ is $\{\xi_{\ell_x - 1}, \xi_{\ell_x - 2}, \cdots, \xi_1, \xi_0\}$
q, $\mathbb{Z}_q$	an integer ring $\mathbb{Z}_q := [0, q-1]$ with the modulus q
$+, -, \cdot$	addition, subtraction, and multiplication in $\mathbb{Z}_q$
ℓ, ℓ_x	$\ell := \log_2 q$; ℓ_x is the precision of the input
λ	$\xi_{\lambda-1}$ is the most significant non-zero bit of ξ; λ is defined as the effective bit length of ξ
r, t	a random mask, a random flipping bit
P_0, P_1, P_2	three participants
$[x]$, $[x]_0$, $[x]_1$	$[x] := ([x]_0, [x]_1)$ is the two-party secret sharing of x
a, b, c, d, e	a Beaver triple with $c = ab$, $d = x - a$, $e = y - b$
$\{u_i\}, \{v_i\}, \{w_i\}$	arrays used in the DReLU protocol
$\Pi\{\cdot\}$	a random shuffle acting on an array
$\vec{\boldsymbol{\phi}}, \vec{\boldsymbol{\psi}}, \vec{\boldsymbol{\theta}}$	vectors used in the MAX/MIN/SORT/MED protocols
$\boldsymbol{M} := \{m_{i,j}\}$	$m_{i,j}$ denotes the (i,j)-th element in the matrix $\boldsymbol{M}$
x_i, θ_i	x_i or θ_i is the i-th item of $\{x_i\}$ or $\vec{\boldsymbol{\theta}}$
$[\{x_i\}]$, $[\vec{\boldsymbol{\theta}}]$	the shares of the array $\{x_i\}$ and the vector $\vec{\boldsymbol{\theta}}$, resp.
α, β, γ	constants

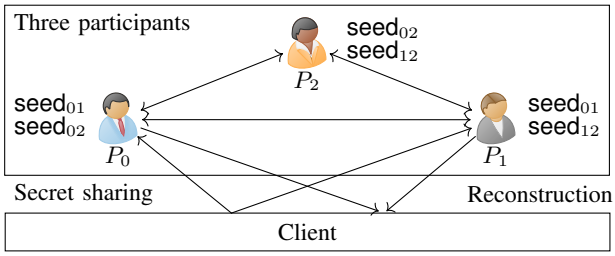


Figure 2: System settings.

2. Preliminary

In this section, we first introduce the system settings in Sect. 2.1, including the topology, the security model, the definition of communication rounds, and the format of numbers used in this system. In Sect. 2.2 we introduce the background of secret sharing. In Sect. 2.3 we recall the truncation protocol proposed in SecureML [24]. All notations used in this paper are listed in Tab. 2.

2.1. System settings

A typical three-party computation (3PC) setting (Fig. 2) is used in this paper, which is also commonly used in previous 3PC PPML works [10], [11], [12], [13], [14], [15], [16], [17], [18], [19]. The input(s) from a client is secretly shared between the participants. Then, the participants securely compute the corresponding shares of the result. Finally, the client reconstructs the shares of the result to form the output(s).

Security model. In our protocol, all three participants (P_0, P_1, and P_2) are static (i.e., non-adaptive) and honest-but-curious, i.e., semi-honest. We take the honest majority setting. It is assumed that there would be no collusion between any two of three participants. Hence, the protocol guarantees none of the participants can break the input, intermediate, or output secrecy alone.

The number of communication rounds. [4] In a 2PC or MPC protocol, some communication passes can be executed in parallel, and thus the round complexity of our protocol refers to the number of unparallelizable rounds, following the previous MPC-based PPML works [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21].

Pre-shared seed. We assume that there are pre-shared pseudorandom seeds among participants, i.e., P_0 and P_1, P_0 and P_2, and P_1 and P_2, share the seeds seed_{01}, seed_{02}, and seed_{12}, respectively (as depicted in Fig. 2). Note that the seeds should be kept secret from the other participant, e.g., P_2 does not know the value of seed_{01}.

Complement Expression. In a modern computer, a number is expressed in the complement form. For a positive number, the computer stores it in its original form; for a negative number, the complement code is used for storage. In this paper, we consider an integer ring $\mathbb{Z}_q = [0, q-1]$, in which q is the modulus and the length of q is $\ell := \log_2 q$. Unless explicitly stated, all arithmetic operations in this paper are assumed in $\mathbb{Z}_q$.

For a number x of length ℓ_x in this ring, i.e., $x \in [0, 2^{\ell_x}) \bigcup (q - 2^{\ell_x}, q)$, it follows the constraint $\ell > \ell_x$. Take $q = 2^{16}, \ell_x = 8$ as an example. The positive integer $x = 0b10111011$ is expressed as $0b0000000010111011$. The negation of x is expressed as $q - x$, which is $0b1111111101010101$ in its binary form. For simplicity, we use integers to illustrate our protocols. A fixed-point computation method is introduced in App. A to handle decimal arithmetic.

Non-linear Functions in PPML. A function satisfying $F(x) = \alpha \cdot x + \beta$ is called a linear function. Otherwise, it is a non-linear function. Convolution is a typical example of linear functions. The common non-linear function used in machine learning is ReLU (Eq. 1), which could be derived from DReLU (Eq. 2). The definitions for other non-linear functions are presented in App. B.

$$\mathrm{ReLU}(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases} = \mathrm{DReLU}(x) \cdot x \tag{1}$$

$$\mathrm{DReLU}(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases} \tag{2}$$

2.2. Secret Sharing

This paper focuses on the additive secret sharing scheme with an unbalanced setting. A plaintext message x is shared between participants P_0 and P_1, which satisfies the relation $x := [x]_0 + [x]_1$. An unbalancing model means that P_2 does not hold any share of the input. Moreover, the shares have the linear homomorphic property, i.e., $[x] + \mathsf{constant} = [x + \mathsf{constant}]$, $[x_1] + [x_2] = [x_1 + x_2]$ and $\mathsf{constant} \cdot [x] = [\mathsf{constant} \cdot x]$. [5] If x and $\mathsf{constant}$ are both binary numbers, $[x] \oplus \mathsf{constant}$ can be computed by $[x] + \mathsf{constant} - 2 \cdot \mathsf{constant} \cdot [x]$. Similarly, $[x_1] \oplus [x_2] = [x_1] + [x_2] - 2 \cdot [x_1 \cdot x_2]$ if both x_1 and x_2 are binary.

In order to perform multiplication among two shares, i.e., $[x \cdot y]$, Beaver triples [4] (the shares of three correlated secrets $[a]$, $[b]$, and $[c]$) are utilized to decrease the online communication overhead. Participants first compute $[d] := [x - a]$ and $[e] := [y - b]$, then reconstruct d and e. Next, the shares $[xy]$ could be locally obtained by $[x \cdot y] = d \cdot e + d \cdot [b] + e \cdot [a] + [c]$. [6]

To further decrease the triple generation overhead, we use a trick proposed in [10]. For more details, P_0 and P_2 generate $[a]_0$, $[b]_0$, and $[c]_0$ using seed_{02}, while P_1 and P_2 generate $[a]_1$ and $[b]_1$ using seed_{12}. Finally, P_2 computes a qualified $[c]_1 = ([a]_0 + [a]_1) \cdot ([b]_0 + [b]_1) - [c]_0$ and sends $[c]_1$ to P_1, which is the only communication overhead during the triple generation. This generation could be mirrored, in which P_2 computes $[c]_0$ and sends $[c]_0$ to P_0, correspondingly. If more than one triple is used in a protocol, the generation of half of the triples could be mirrored to balance the communication.

The seeds are also utilized to simplify the share generations for some special values. For example, when there is a need to have shares of $\mathsf{constant}$ among P_0 and P_1. P_0 assigns a generated random ring element r from seed_{01} as $[\mathsf{constant}]_0$, while P_1 regards $\mathsf{constant} - r$ as $[\mathsf{constant}]_1$. The zero-value and one-value secure shares would be utilized in our protocols.

2.3. The Truncation Protocol with Errors

SecureML [24] proposes an MPC-based truncation protocol to keep the precision of a secretly shared fixed-point

4. For a traditional two-party security protocol, a two-pass protocol is named as an "one-round" protocol. We do not use this definition in this paper.

5. For $[x] + \mathsf{constant}$, only one participant is required to add the constant to his secret share, e.g., $[x]_0 + \mathsf{constant}$, while the other participant does nothing. Such notation is used to represent adding a constant number to a secret share for the rest of the paper.

6. The equation lies in $x \cdot y = (x - a + a) \cdot (y - b + b) = (d + a)(e + b) = (de + db + ea + ab) = (de + db + ea + c)$.

number during the computations. In this non-interactive protocol, P_0 and P_1 (respectively holding the $[x]_0$ and $[x]_1$ of a shared x) perform their own right shifting operation(s) on their shares individually. The k-bit non-cyclic right shifting is denoted by $\text{rShift}(x,k)$, without padding zero in the left hand. Specifically, P_0 directly right shifts its share for k bits. P_1 takes the input negation and then does another negation after k-bit shifting. Finally, P_0 and P_1 withhold the shares

$$[\text{TRC}(x,k)]_0 := \text{rShift}([x]_0,k),$$
$$[\text{TRC}(x,k)]_1 := q - \text{rShift}(q-[x]_1,k).$$

Due to the one-bit error, the k-bit truncation protocol for x (denoted by $\text{TRC}(x,k)$) would lose one bit of accuracy at the least significant bit. We define the output of an exact truncation as that of k-bit right shifting, i.e., $\mathsf{trc} := \text{rShift}(x,k)$. Hence, it is possible that

$$[\text{TRC}(x,k)]_0 + [\text{TRC}(x,k)]_1 = \text{TRC}(x,k) \in \{\mathsf{trc}-1, \mathsf{trc}, \mathsf{trc}+1\}.$$

For an ℓ_x-precision input $x \in \mathbb{Z}_q$ and $\ell = \log_2 q$, let $\xi = \xi(x) := x$ if $x \in [0, 2^{\ell_x})$ (a defined positive input and a zero input) and $\xi = \xi(x) := q - x$ if $x \in (q-2^{\ell_x}, q)$ (a defined negative input). Hence, ξ keeps as defined positive. Then, the truncation result with errors could be presented by Lemma 1.

Lemma 1. In a ring $\mathbb{Z}_q$, let $x \in [0,2^{\ell_x}) \bigcup (q-2^{\ell_x}, q)$, where $\ell > \ell_x + 1$. Then we have the following results with probability $1 - 2^{\ell_x+1-\ell}$:

- If $x \in [0, 2^{\ell_x})$, then $\text{TRC}(x,k) = \text{rShift}(\xi,k) + \mathsf{bit}$, where $\mathsf{bit} = 0$ or 1.
- If $x \in (q-2^{\ell_x}, q)$, then $\text{TRC}(x,k) = q - \text{rShift}(\xi,k) - \mathsf{bit}$, where $\mathsf{bit} = 0$ or 1.

Lemma 1 proves the corresponding statement proposed in [24, Sect. 4.1] with a more precise characterization of the ± 1 error. We show the possible truncation error is $+1$ or -1 for a positive or a negative input respectively, while [24, Theorem 1] simply states the existence of potential ± 1 error. Moreover, Lemma 2 presents a special case of Lemma 1, i.e., when errors do occur. As we will show later, our sign determination protocol benefits from a better understanding of how and when the ± 1 error occurs. The proofs of Lemma 1 and Lemma 2 are presented in App. C.

3. Two-round DReLU Protocol without Preprocessing

In this section, we present the basis of Bicoptor, the sign determination (DReLU) protocol. We first describe how do we determine the sign of an input by using repeated truncations in Sect. 3.1. In Sect. 3.2, we utilize the result array from repeated truncations to form a "strawman" protocol which computes the DReLU function, but suffers from a few privacy issues. Finally, in Sect. 3.3 we show that the privacy issues can be solved by adding additional improvements on top of the strawman protocol, which leads to the complete DReLU protocol.

TABLE 3: An example of truncations **without** errors.

	$x = 0b00010110$	$x = q - 0b00010110$
Opt.	**Value**	**Value**
TRC(x,1)	$0b00001011$	$q - 0b00001011$
TRC(x,2)	$0b00000101$	$q - 0b00000101$
TRC(x,3)	$0b00000010$	$q - 0b00000010$
TRC(x,4)	$\mathbf{0b00000001}$	$\mathbf{q - 0b00000001}$
TRC(x,5)	$0b00000000$	$0b00000000$
TRC(x,6)	$0b00000000$	$0b00000000$
TRC(x,7)	$0b00000000$	$0b00000000$
TRC(x,8)	$0b00000000$	$0b00000000$

3.1. The necessity of repeated truncations

Recall that our aim is to determine the output of $\text{TRC}(x,\lambda-1)$ or $\text{TRC}(x,\lambda)$, where λ is the effective bit-length of ξ. Lemma 3 proves that if $\text{TRC}(x,\lambda-1)$ or $\text{TRC}(x,\lambda)$ is 1, the input is positive; if it is $q-1$, the input is negative. However, as mentioned before, λ is unknown to all participants in an MPC context. To address this problem, we can simply perform ℓ_x times of probabilistic truncations and output an array $\{\text{TRC}(x,1), \cdots, \text{TRC}(x,\lambda-1), \text{TRC}(x,\lambda), \cdots, \text{TRC}(x,\ell_x)\}$. Lemma 4 states that the result of $\text{TRC}(x,\lambda-1)$ and $\text{TRC}(x,\lambda)$ is included in this array. In other words, by checking the existence of 1 or $q-1$ in this array, we can determine the sign of the input.

Lemma 5 and Lemma 6 prove that the behavior of the tail elements in the array follows a specific pattern. Based on Lemma 1-6, we formally introduce Theorem 1.

Theorem 1. For an ℓ_x-bits input $x \in \mathbb{Z}_q$, let $\xi = \xi(x) := x$ if $x \in (0, 2^{\ell_x})$, and let $\xi = \xi(x) := q-x$ if $x \in (q-2^{\ell_x}, q)$, in which $\ell := \log_2 q$. The binary form ξ is defined as $\{\xi_{\ell_x-1}, \xi_{\ell_x-2}, \cdots, \xi_1, \xi_0\}$, in which ξ_i denotes the i-th bit and $\xi := \sum_{i=0}^{\ell_x-1} \xi_i \cdot 2^i$. λ is the effective bit length of ξ, i.e., $\xi_{\lambda-1} = 1$ and $\lambda+1 < \ell$. Set $\xi := \xi'' \cdot 2^k + \xi'$, where $\xi'' \in [0, 2^{\ell_x-k})$ and $\xi' \in [0, 2^k)$, so that $\text{rShift}(\xi,k) = \xi''$. Then, for any value $\hat{\ell} \geq \lambda$, we have the following results with probability $1 - 2^{\lambda+1-\hat{\ell}}$:

- For $x = \xi$, there exists positive numbers λ' and λ'' $(\lambda' \leq \lambda'' \leq \ell_x)$ satisfying $\text{TRC}(\xi,j) = 1$ for $\lambda' \leq j \leq \lambda''$, and $\text{TRC}(\xi,j) = 0$ for $j > \lambda''$.
- For $x = q - \xi$, there exists positive numbers λ' and λ'' $(\lambda' \leq \lambda'' \leq \ell_x)$ satisfying $\text{TRC}(q-\xi,j) = q-1$ for $\lambda' \leq j \leq \lambda''$, and $\text{TRC}(q-\xi,j) = 0$ for $j > \lambda''$.

An example of the outcome array with ℓ_x times of exact truncations (without errors) for both a positive and a negative input is shown in Tab. 3. Tab. 4 shows an example of the outcome array with ℓ_x times of probabilistic truncations (with errors). Due to the limited space, we postpone the formal proof to App. C.

3.2. Strawman DReLU protocol

After outputting an array $\{\text{TRC}(x,1), \cdots, \text{TRC}(x,\ell_x)\}$, we are now ready to produce the "strawman" protocol that

TABLE 4: An example of truncations **with** errors.

	$x = 0b00010110$		$x = q - 0b00010110$	
Opt.	**Value**	**Err.**	**Value**	**Err.**
TRC(x,1)	$0b00001011$	0	$q - 0b00001011$	0
TRC(x,2)	$0b00000110$	$+1$	$q - 0b00000110$	-1
TRC(x,3)	$0b00000010$	0	$q - 0b00000010$	0
TRC(x,4)	**0b00000001**	0	**q−0b00000001**	0
TRC(x,5)	**0b00000001**	$+1$	**q−0b00000001**	-1
TRC(x,6)	**0b00000001**	$+1$	**q−0b00000001**	-1
TRC(x,7)	$0b00000000$	0	$0b00000000$	0
TRC(x,8)	$0b00000000$	0	$0b00000000$	0

TABLE 5: Subtracting one from each element in the truncation result array (with errors).

	$x = 0b00010110$		$x = q - 0b00010110$	
Opt.	**Value**	**Err.**	**Value**	**Err.**
$\text{TRC}(x,1) - 1$	$0b00001010$	0	$q - 0b00001010$	0
$\text{TRC}(x,2) - 1$	$0b00000101$	$+1$	$q - 0b00000101$	-1
$\text{TRC}(x,3) - 1$	$0b00000001$	0	$q - 0b00000001$	0
$\text{TRC}(x,4) - 1$	**0b00000000**	0	$q - 0b00000010$	0
$\text{TRC}(x,5) - 1$	**0b00000000**	$+1$	$q - 0b00000010$	-1
$\text{TRC}(x,6) - 1$	**0b00000000**	$+1$	$q - 0b00000010$	-1
$\text{TRC}(x,7) - 1$	$q - 0b00000001$	0	$q - 0b00000001$	0
$\text{TRC}(x,8) - 1$	$q - 0b00000001$	0	$q - 0b00000001$	0

Algorithm 1 Strawman DReLU protocol.

Input: the shares of x
Output: the shares of $\text{DReLU}(x)$
// P_0 *and* P_1 *initialization.*
1: P_0 and P_1 generate ℓ_x numbers of non-zero random ring elements $\{r_1, \cdots, r_{\ell_x}\}$ from seed_{01}.
2: P_0 and P_1 set $[u_i] := [\text{TRC}(x, i)] - 1$ and $[v_i] := r_i \cdot [u_i]$, for $\forall i \in [1, \ell_x]$.
3: P_0 and P_1 set $[\{w_i\}] := [\Pi\{v_i\}]$, using the shuffle-seed generated from seed_{01}.
4: P_0 and P_1 send the shares $[\{w_i\}]$ to P_2.
// P_2 *processes.*
5: P_2 reconstructs $\{w_i\}$, and sets $\text{DReLu}(x) = 1$ if there exists zero(s) in the array; otherwise $\text{DReLu}(x) = 0$.
6: P_2 shares $\text{DReLu}(x)$ to P_0 and P_1.

computes the DReLU function. According to the secrecy requirement in our system, P_2 should not learn anything about the input x. Hence, P_0 and P_1 should mask and shuffle the shares of this array before revealing to P_2. The masking process could be achieved by multiplying each element in this array by a non-zero random ring element. The random ring elements and the shuffle-seed could be generated from a shared seed between P_0 and P_1, i.e., seed_{01}.

We immediately notice that, we are not able to determine the target element after multiplying 1 or $q - 1$ by a non-zero random ring element. To address this issue, we can either add one to or subtract one from each element in this array according to the needs. For simplicity, we use $\{u_i\}$ to represent the new array (Tab. 5). By doing this, checking the existence of 1 or $q - 1$ in the original array is transferred into checking that of 0 in $\{u_i\}$, which would not be affected by the masking process. Now, P_2 can reconstruct the shares from P_0, P_1 and output the result of the DReLU function according to the existence of 0. We present the strawman DReLU protocol in Alg. 1 and Fig. 3a. However, the strawman protocol still suffers from a few privacy issues.

- Despite multiplicative masking and shuffling, P_2 still learns the sign of the DReLU result which is not allowed according to our security definition.
- Due to the one-bit error, there might be continuous zeros in the array $\{u_i\}$ (Tab. 5), which leaks information about the range of x.
- We omit the corner case in the strawman DReLU protocol. In particular, the protocol in Alg. 1 does not work when $x = 0$ or $x = 1$.

In the next subsection, we show how to solve these problems using common techniques in secure multiparty computation.

3.3. The DReLU procotol

In this subsection, we improve the strawman protocol to form the complete DReLU protocol.

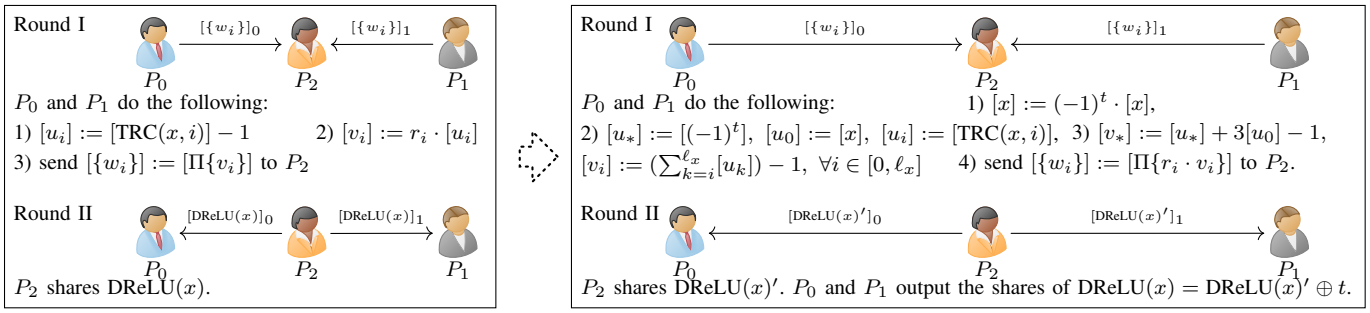
Masking the DReLU output with random input negation. We can prevent P_2 from learning the actual sign of $x := (-1)^t \cdot x$ by flipping a coin t using the shared seed_{01} between P_0 and P_1. After P_2 responds $\text{DReLU}(x)'$, P_0 and P_1 negate $\text{DReLU}(x)'$ according to t. This negation $\text{DReLU}(x) = t \oplus \text{DReLU}(x)' = t + \text{DReLU}(x)' - 2t \cdot \text{DReLU}(x)'$ is a linear operation since P_0 and P_1 knows t. Since P_2 does not know about seed_{01}, $\text{DReLU}(x)$ is completely masked by t.

Dealing with continuous zeros using summation. To avoid the possible continuous zeros in the output array leaking information about the range of the input x, P_0 and P_1 could instead send the summation of subarrays, i.e., let $u_i := \text{TRC}([x], i)$ and $v_i := (\sum_{k=i}^{\ell_x} u_k) - 1$, for $i \in [1, \ell_x]$. By doing this, the continuous 1-values in $\{u_i\}$ would result in a single 0 in $\{v_i\}$ for a positive input. Tab. 6 depicts an example.

Handling the input of one. Notice that if the input value is 1, then truncating it even by one position would result in zero instead of the 1 or $q-1$ that we desire. Thus we append $[x]$ itself to the beginning of the output array $\{u_i\}$. This step is applied after the previous random negation operation.

Handling the input of zero. The DReLU function is defined to output 1 at the input point of 0, i.e., $\text{DReLU}(0) := 1$. To accomplish this goal, we append an additional item to the truncation result array, denoted as u_*. In particular, we set $u_* := (-1)^t$ and $v_* := u_* + \mathsf{coeff} \cdot u_0 - 1$ where u_0 should be $(-1)^t x$ (after the random negation). coeff is a positive constant. We want v_* to satisfy the following constraint

$$\text{if and only if } x = 0, t = 0, \text{ then } v_* = 0.$$



(a) Strawman-DReLU (b) DReLU

Figure 3: Strawman-DReLU and DReLU protocol overview.

This additional element v_* assures the correct output of DReLU when $x = 0$, regarding the flipping coin t, meanwhile, it does not affect the correctness of DReLU other inputs.

We select the value of coeff as 3 and briefly discuss the reason to do so. Apparently when $x = 0$ the coeff term does not contribute to v_* and thus we only need to concentrate on the $x \neq 0$ case, which should satisfy $(-1)^t(1 + \mathsf{coeff} \cdot x) \neq -1 \pmod q$. An intuitive way could be used to find the minimum coeff value, if the input x ranges $(0, 2^{\ell_x}) \bigcup (q - 2^{\ell_x}, q)$. When $\mathsf{coeff} = 1$ or 2, $x = q-2$ or $x = q-1$ breaks the aforementioned constraint, respectively. When $\mathsf{coeff} = 3$, suppose the condition does not hold, which implies $\mathsf{coeff} \cdot x = 0$ or $-2 \pmod q$. Since we have $\ell_x \leq \ell = \log_2 q$, it holds that $\mathsf{coeff} \cdot x < q$ if $x > 0$ and $\mathsf{coeff} \cdot x > -q$ if $x < 0$. Therefore the constraint that $\mathsf{coeff} \cdot x \neq 0$ and $\neq -2$ is satisfied.

Finally, these improvements are combined with our complete DReLU protocol (Alg. 2 and Fig. 3b). P_0 and P_1 first generate a random bit t from seed_{01}, set negate the input accordingly, i.e., $x := (-1)^t \cdot x$. They also set u_* and u_0 to deal with the special input zero and one. Then they execute a repeated times of probabilistic truncations on the input to get an array $\{u_i\}$, in which $u_i := \mathrm{TRC}(x,i), \forall i \in [1,\ell_x]$. Next, v_* is set to $u_* + 3u_0 - 1$, and $v_i = (\sum_{k=i}^{\ell_x} u_k) - 1, \forall i \in [0,\ell_x]$. Recall the summation would eliminate the possible continuous zeros that leak input information. P_0 and P_1 finally mask and shuffle the output array using seed_{01}, i.e., $\{w_i\} := \Pi\{r_i \cdot v_i\}$. After P_2 reconstructs $\{w_i\}$ and sets $\mathrm{DReLU}(x)'$ according to whether $\{w_i\}$ contains zero, P_2 shares the output to P_1 and P_0 who subsequently removes the random negation to get shares of $\mathrm{DReLU}(x)$. We demonstrate the processing of a positive input example in Fig. 7 in App. D.

TABLE 6: Summing-then-subtracting-one on the truncation results to avoid the leakage of the input range, **with** one-bit errors.

	$x = 0b00010110$	$x = q - 0b00010110$
Opt.	**Value**	**Value**
$\sum_{k=1}^{8} \mathrm{TRC}(x,k) - 1$	$0b00010101$	$q - 0b00010111$
$\sum_{k=2}^{8} \mathrm{TRC}(x,k) - 1$	$0b00001010$	$q - 0b00001100$
$\sum_{k=3}^{8} \mathrm{TRC}(x,k) - 1$	$0b00000100$	$q - 0b00000110$
$\sum_{k=4}^{8} \mathrm{TRC}(x,k) - 1$	$0b00000010$	$q - 0b00000100$
$\sum_{k=5}^{8} \mathrm{TRC}(x,k) - 1$	$0b00000001$	$q - 0b00000011$
$\sum_{k=6}^{8} \mathrm{TRC}(x,k) - 1$	**0b00000000**	$q - 0b00000010$
$\sum_{k=7}^{8} \mathrm{TRC}(x,k) - 1$	$q - 0b00000001$	$q - 0b00000001$
$\sum_{k=8}^{8} \mathrm{TRC}(x,k) - 1$	$q - 0b00000001$	$q - 0b00000001$

Algorithm 2 DReLU protocol.

Input: the shares of x
Output: the shares of $\mathrm{DReLU}(x)$
// P_0 and P_1 initialization.
1: P_0 and P_1 generate $\ell_x + 2$ numbers of non-zero random ring elements $\{r_*, r_0, r_1, \cdots, r_{\ell_x}\}$ from seed_{01}.
2: P_0 and P_1 set $[x] := (-1)^t \cdot [x]$.
3: P_0 and P_1 set $[u_*] := [(-1)^t]$, $[u_0] := [x]$, and $[u_i] := [\mathrm{TRC}(x,i)]$, $\forall i \in [1,\ell_x]$.
4: P_0 and P_1 set $[v_*] := [u_*] + 3 \cdot [u_0] - 1$, and $[v_i] := (\sum_{k=i}^{\ell_x}[u_k]) - 1, \forall i \in [0,\ell_x]$.
5: P_0 and P_1 set $[\{w_i\}] := [\Pi\{r_i \cdot v_i\}]$, using the shuffle-seed generated from seed_{01}.
6: P_0 and P_1 send the shares $[\{w_i\}]$ to P_2.
// P_2 processes.
7: P_2 reconstructs $\{w_i\}$ and sets $\mathrm{DReLu}(x)' = 1$ if there exists zero(s) in $\{w_i\}$; otherwise $\mathrm{DReLu}(x)' = 0$.
8: P_2 shares $\mathrm{DReLu}(x)'$ to P_0 and P_1.
9: P_0 and P_1 output the shares of $t \oplus \mathrm{DReLu}(x)'$.

4. Protocols for Other Non-linear Functions

In this section, we describe how the DReLU protocol could be extended to securely compute various non-linear functions including the Equality, ABS, ReLU, Dynamic ReLU (Leaky ReLU, PReLU, RReLU), ReLU6, Piecewise Linear Unit (PLU), MAX, MIN, SORT, and median (MED) functions. The definition of these functions is presented in App. B.

4.1. DReLU Variants

The protocols for the functions $\mathrm{BitExt}(x)$, $\mathrm{MSB}(x)$ and $\mathrm{CMP}(x,y)$ could be derived from the DReLU function (recall that we assume the inputs are encoded in the com-

Algorithm 3 Equality protocol.

Input: the shares of x and y
Output: the shares of Equality(x, y)

1: P_0 and P_1 set $[x-y] = [x]-[y]$ and $[y-x] = [y]-[x]$ as the input shares of the first and second DReLU instance, using the random bits t_0 and t_1, respectively.
2: P_2 obtains DReLU$(x-y)'$ and DReLU$(y-x)'$ from the two instances, sets DReLU$'' :=$ DReLU$(x-y)' \oplus$ DReLU$(y-x)'$, and shares DReLU$''$ to P_0 and P_1.
3: P_0 and P_1 set $t' := t_0 \oplus t_1$, and output the shares of $1 - t' \oplus$ DReLU$''$.

Algorithm 4 ReLU protocol.

Input: the shares of x
Output: the shares of ReLU(x)

1: P_0 and P_2 generate $[a]_0, [b]_0, [c]_0$ from seed_{02}. P_1 and P_2 generate $[a]_1, [b]_1$ from seed_{12}
2: P_0 and P_1 input $[x]$ to DReLU.
3: P_0 sends $[d]_0 := [x]_0 - [a]_0$ to P_1, and P_1 sends $[d]_1 := [x]_1 - [a]_1$ to P_0.
4: P_2 obtains DReLU$(x)'$ from DReLU, computes $[c]_1 := ([a]_0 + [a]_1) \cdot ([b]_0 + [b]_1) - [c]_0$, and $e :=$ DReLU$(x)' - ([b]_0 + [b]_1)$; and sends e to P_0, e and $[c]_1$ to P_1.
5: P_0 and P_1 set $d = [d]_0 + [d]_1$, and computes the shares of the output, i.e., $[\text{ReLU}(x)] := (1-2t) \cdot (d \cdot e + d[b] + e[a] + [c]) + t \cdot [x]$.

plementary form as in Sect. 2). We mainly introduce how to construct the Equality function in Alg. 3, which invokes the DReLU protocol (Alg. 2) twice. For inputs x and y, we have

$$\begin{aligned}&\text{Equality}(x, y) = 1 - \text{DReLU}(x - y) \oplus \text{DReLU}(y - x)\\ =&1 - (\text{DReLU}(x - y)' \oplus t_0) \oplus (\text{DReLU}(y - x)' \oplus t_1)\\ =&1 - \text{DReLU}'' \oplus t',\end{aligned}$$

in which t_0 and t_1 are the random flipping bits in the two invocations, respectively. Specifically, after P_0 and P_1 invoke DReLU$(x-y)$ and DReLU$(y-x)$, the response from P_2 can be computed by an XOR operation, i.e., the shares of DReLU$'' :=$ DReLU$(x-y)' \oplus$ DReLU$(y-x)'$. Finally, P_0 and P_1 also XOR the two random bits, i.e., $t' := t_0 \oplus t_1$, then obtain the shares of the Equality function output.

4.2. ReLU

As mentioned in Sect. 2.1, the ReLU function could be decomposed into DReLU multiplied by the input value. Trivially, P_0 and P_1 could utilize the Beaver triple to perform this multiplication on shared values, as shown in Fig. 4a. Nevertheless, when the triple is generated by P_2, the communication for DReLU and triple generation could be integrated to further reduce the round number from three to two, as shown in Fig. 4b. In particular, notice that after completing the DReLU protocol, P_2 holds DReLU$(x)'$, and DRelu$(x) =$ DRelu$(x)' = t +$ DRelu$(x)' - 2 \cdot t \cdot$ DRelu$(x)'$. Moreover since ReLU$(x) = x \cdot$ DReLU(x), we have

$$\begin{aligned}&[\text{ReLU}(x)] = (1-2t) \cdot [x \cdot \text{DRelu}(x)'] + t \cdot [x]\\ =&(1-2t) \cdot (d \cdot e + d[b] + e[a] + [c]) + t \cdot [x].\end{aligned}$$

The share multiplication is transformed from $[\text{DRelu}(x) \cdot x]$ to $[\text{DRelu}(x)' \cdot x]$, which is the key to the communication optimization. It is worth noticing that $e :=$ DRelu$(x)' - b$ does not break the secrecy of DRelu$(x)'$, since none of P_0 and P_1 knows the integrate b value. In this way, we manage to decrease the communication rounds required for the protocol down to two. This process is shown in Alg. 4

4.3. ABS, Dynamic ReLU, MAX2, MIN2, and Funnel ReLU

Building on the ReLU protocol, the protocols for the ABS (Eq. 4), Dynamic Relu (Eq. 5), MAX2 (Eq. 10), MIN2 (Eq. 11), and Funnel ReLU (Eq. 12) functions could be constructed as follows.

ABS. Since ABS$(x) = (2 \cdot \text{DReLU}(x) - 1) \cdot x$,

$$\begin{aligned}&[\text{ABS}(x)] = [(2 \cdot \text{DReLU}(x) - 1) \cdot x]\\ =&[(2 \cdot (\text{DReLU}(x)' \oplus t) - 1) \cdot x]\\ =&[(2 \cdot (\text{DReLU}(x)' + t - 2 \cdot \text{DReLU}(x)' \cdot t) - 1) \cdot x]\\ =&(2-4t) \cdot [x \cdot \text{DReLU}(x)'] + (2t-1) \cdot [x].\end{aligned}$$

Hence, P_0 and P_1 would computes

$$(2-4t) \cdot (de + d[b] + e[a] + c) + (2t-1) \cdot [x]$$

in the step 5 of Alg. 4 for the ABS function.

Dynamic ReLU. Similarly, P_0 and P_1 would computes

$$\begin{aligned}&(1-2t) \cdot (\alpha_1 - \alpha_0) \cdot (de + d[b] + e[a] + c)\\ +&((\alpha_1 - \alpha_0)t + \alpha_0) \cdot [x]\end{aligned}$$

in the step 5 of Alg. 4 for the Dynamic ReLU function.

MAX2 and MIN2. Moreover, for MAX2(x, y) and MIN2(x, y), P_0 and P_1 would reconstruct $d := (x - y) - a$ or $d := (y - x) - a$ in the step 3, then compute

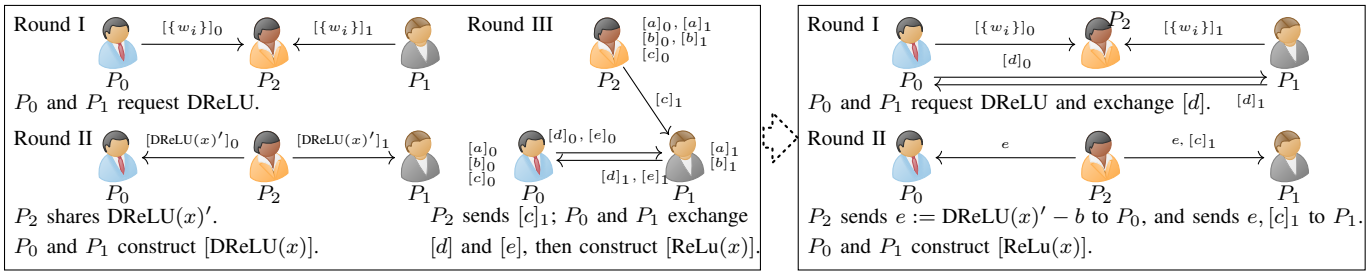
$$\begin{aligned}&(1-2t) \cdot (de + d[b] + e[a] + c) + t \cdot [x - y] + [y],\\ \text{or }&(1-2t) \cdot (de + d[b] + e[a] + c) + t \cdot [y - x] + [x]\end{aligned}$$

in the step 5 of Alg. 4, respectively.

Funnel ReLU. For the Funnel ReLU function, the reconstructed d should be $(x - \text{T}(x)) - a$ in the step 3, and the computation in the step 5 of Alg. 4 is

$$(1-2t) \cdot (de + d[b] + e[a] + c) + t \cdot [x - \text{T}(x)] + [\text{T}(x)].$$

Since we only modify the message contents used in the two-round ReLU protocol in Alg. 4, all the protocols described here keep the round complexity of two.



(a) Strawman-ReLU (b) ReLU

Figure 4: Strawman-ReLU and ReLU protocol overview.

4.4. Piecewise Linear Unit (PLU)

The ReLU and other variant functions we considered above could be viewed as piecewise functions with two segments, which involve one DReLU. More generally, the number of DReLU invocations is equal to the number of segments in a piecewise function minus one. Moreover, for $m+2$ segments in a PLU function (Eq. 8), there would be $m+1$ interval points (i.e., $\gamma_j, \forall j \in [0,m]$) and $m+2$ linear function coefficients (i.e., $\alpha_j, \beta_j, \forall j \in [0, m+1]$), there would be $m+1$ DReLU invocations and $m+2$ multiplications.

For the j-th segment, the corresponding monomial is

$$\begin{aligned}
&(\mathrm{DReLU}(x-\gamma_{j-1}) \oplus \mathrm{DReLU}(x-\gamma_j)) \cdot (\alpha_j \cdot x + \beta_j) \\
=&(t_{j-1} \oplus \mathrm{DReLU}(x-\gamma_{j-1})' \oplus t_j \oplus \mathrm{DReLU}(x-\gamma_j)') \\
&\cdot (\alpha_j \cdot x + \beta_j) \\
=&(t'_{j-1,j} \oplus \mathrm{DReLU}''_{j-1,j}) \cdot (\alpha_j \cdot x + \beta_j) \\
=&\alpha_j \cdot (1-2t'_{j-1,j}) \cdot \mathrm{DReLU}''_{j-1,j} \cdot x + \alpha_j \cdot t'_{j-1,j} \cdot x \\
&+ \beta_j \cdot (1-2t'_{j-1,j}) \cdot \mathrm{DReLU}''_{j-1,j} + \beta_j \cdot t'_{j-1,j},
\end{aligned}$$

where P_0 and P_1 computes $t'_{j-1,j} := t_{j-1} \oplus t_j$ and P_2 computes $\mathrm{DReLU}''_{j-1,j} := \mathrm{DReLU}(x-\gamma_{j-1})' \oplus \mathrm{DReLU}(x-\gamma_j)'$. The two special end-point segments (the m-th and 0-th ones) are as follows.

$$\begin{aligned}
&(\mathrm{DReLU}(x-\gamma_m) \oplus 0) \cdot (\alpha_{m+1} \cdot x + \beta_{m+1}) \\
=&\mathrm{DReLU}(x-\gamma_m) \cdot (\alpha_{m+1} \cdot x + \beta_{m+1}) \\
=&(t_m \oplus \mathrm{DReLU}(x-\gamma_m)') \cdot (\alpha_{m+1} \cdot x + \beta_{m+1}) \\
=&\alpha_{m+1} \cdot (1-2t_m) \cdot \mathrm{DReLU}(x-\gamma_m)' \cdot x + \\
&+ \beta_{m+1} \cdot (1-2t_m) \cdot \mathrm{DReLU}(x-\gamma_m)' + \\
&+ \alpha_{m+1} \cdot t_m \cdot x + \beta_{m+1} \cdot t_m,
\end{aligned}$$

and

$$\begin{aligned}
&(1 \oplus \mathrm{DReLU}(x-\gamma_0)) \cdot (\alpha_0 \cdot x + \beta_0) \\
=&(1 \oplus t_0 \oplus \mathrm{DReLU}(x-\gamma_0)') \cdot (\alpha_0 \cdot x + \beta_0) \\
=&\alpha_0 \cdot (2t_0-1) \cdot \mathrm{DReLU}(x-\gamma_0)' \cdot x + \alpha_0 \cdot (1-t_0) \cdot x \\
&+ \beta_0 \cdot (2t_0-1) \cdot \mathrm{DReLU}(x-\gamma_0)' + \beta_0 \cdot (1-t_0).
\end{aligned}$$

The shares of the triples $([a_j], [b_j], [c_j], \forall j \in [0, m+1])$ are used to compute the shares $[\mathrm{DReLu}(x-\gamma_0)' \cdot x]$, $[\mathrm{DReLU}''_{j-1,j} \cdot x]$ $(\forall j \in [1,m])$, and $[\mathrm{DReLu}(x-\gamma_m)' \cdot x]$.

Algorithm 5 PLU protocol.

Input: the shares of x
Output: the shares of $\mathrm{PLU}(x)$

1: P_0 and P_1 generate the shares of $m+2$ numbers of triples from seed_{02} and seed_{12}, respectively, where all a values are identical. Skip the j-th triple if $\alpha_j = 0, \forall j \in [0, m+1]$.
2: P_0 sends $[d]_0 := [x]_0 - [a]_0$ to P_1, and P_1 sends $[d]_1 := [x]_1 - [a]_1$ to P_0.
3: P_0 and P_1 input $[x-\gamma_j]$ to the j-th DReLU instance, $\forall j \in [0,m]$.
4: P_2 obtains $\mathrm{DReLU}(x-\gamma_j)'$ from the j-th DReLU instance, and computes $\mathrm{DReLU}''_{j-1,j} := \mathrm{DReLU}(x-\gamma_{j-1})' \oplus \mathrm{DReLU}(x-\gamma_j)', \forall j \in [0,m]$.
5: P_2 sets $e_0 := \mathrm{DReLU}(x-\gamma_0)' - b_0$, $e_j := \mathrm{DReLU}''_{j-1,j} - b_j$ $(\forall j \in [1,m])$, $e_{m+1} := \mathrm{DReLU}(x-\gamma_m)' - b_{m+1}$.
6: P_2 sends e_j to P_0 and $e_j, [c_j]_1$ to P_1, $\forall j \in [0, m+1]$.
7: P_0 and P_1 reconstruct d, set $t'_{j-1,j} := t_{j-1} \oplus t_j$, $\forall j \in [1,m]$, and output the shares

$$\begin{aligned}
&=\alpha_{m+1} \cdot (1-2t_m) \\
&\quad \cdot (de_{m+1} + d[b_{m+1}] + e_{m+1}[a] + [c_{m+1}]) \\
&\quad + \beta_{m+1} \cdot (1-2t_m) \cdot \mathrm{DReLU}(x-\gamma_m)' \\
&\quad + \alpha_{m+1} \cdot t_m \cdot x + \beta_{m+1} \cdot t_m \\
&+ \ldots \\
&+ \alpha_j \cdot (1-2t'_{j-1,j}) \cdot (de_j + d[b_j] + e_j[a] + [c_j]) \\
&\quad + \beta_j \cdot (1-2t'_{j-1,j}) \cdot \mathrm{DReLU}''_{j-1,j} \\
&\quad + \alpha_j \cdot t'_{j-1,j} \cdot x + \beta_j \cdot t'_{j-1,j} \\
&+ \ldots \\
&+ \alpha_0 \cdot (2t_0-1) \cdot (de_0 + d[b_0] + e_0[a] + [c_0]) \\
&\quad + \beta_0 \cdot (2t_0-1) \cdot \mathrm{DReLU}(x-\gamma_0)' \\
&\quad + \alpha_0 \cdot (1-t_0) \cdot x + \beta_0 \cdot (1-t_0),
\end{aligned}$$

In total, there would be $m+2$ numbers of triples being consumed. Note that the multiplicand x is the same in these $m+2$ numbers of multiplications (if all α_j are not zero), we would use the same a value in these triples, leading to the same $d := x - a$ value. Alg. 5 describes the protocol in detail. For simplicity, we consider a non-mirror case in

Algorithm 6 MAX protocol.

Input: the shares of $\vec{\psi} = (\psi_1, \cdots, \psi_n)$
Output: the shares of MAX($\vec{\psi}$)

1: P_0 and P_1 shuffle the input array $\vec{\psi}$ using the same shuffle-seed generated from seed_{01} and get $\vec{\phi}$, then compare each pair of elements in $\vec{\phi}$ using uCMP, without duplication.
2: P_0, P_2 and P_1, P_2 generate the shares of n triples from seed_{02} and seed_{12}, respectively.
3: P_0 sends $[d_i]_0 := [\phi_i]_0 - [a_i]_0$ to P_1, and P_1 sends $[d_i]_1 := [\phi_i]_1 - [a_i]_1$ to P_0, $\forall i \in [1, n]$.
4: P_0 and P_1 send the requests of uCMP(ϕ_i, ϕ_j) ($\forall i, j \in [1, n]$ and $i \neq j$) to P_2.
5: P_2 determines the target indexes $i^\star$ satisfying $\forall i < i^\star$, uCMP(ϕ_i, ϕ_{i^*}) is 0 or NULL, and $\forall j > i^\star$, uCMP(ϕ_{i^*}, ϕ_j) is 1 or NULL.
6: P_2 constructs a weight-one vector $\vec{\boldsymbol{\theta}}$, in which $\theta_{i^\star} = 1$ and $\theta_i = 0, \forall i \neq i^\star, i \in [1, n]$.
7: P_2 sets $e_i := \theta_i - b_i$, and sends e_i to P_0, e_i and $[c_i]_1$ to P_1, $\forall i \in [1, n]$.
8: P_0 and P_1 set $d_i := [d_i]_0 + [d_i]_1$, $\forall i \in [1, n]$, and compute the shares $[\text{MAX}(\vec{\psi})] = \sum_{i=1}^{n} [\phi_i \cdot \theta_i] = \sum_{i=1}^{n} (d_i \cdot e_i + d_i[b_i] + e_i[a_i] + [c_i])$

Alg. 5. As we discussed in Sect. 2.2, $\lceil \frac{m+2}{2} \rceil$ triples could be generated regularly (i.e., P_2 sends $[c]_1$ to P_1) while the generation of the remained $\lfloor \frac{m+2}{2} \rfloor$ triples could be mirrored (i.e., P_2 sends $[c]_0$ to P_0). We achieve this balancing in our implementation.

ReLU6 is a special case for PLU, in which $m = 1, \alpha_0 = \beta_0 = \beta_1 = \alpha_2 = \gamma_0 = 0, \alpha_1 = 1, \beta_2 = \gamma_2 = 6$.

4.5. MAX

In this section, we present two types of Maxpool protocols, namely, binary-tree-based Maxpool [7] and 2-round Maxpool, each with its own advantages. While the bandwidth is not saturated and the latency is dominating the performance, e.g. under LAN settings with a small batch size for small n, the 2-round Maxpool protocol provides better performance. In contrast, the binary-tree-based Maxpool protocol performs better while the network bandwidth is not rich, e.g. under WAN settings or LAN settings with a large batch size. The communication complexity of the 2-round and binary-tree-based maxpool protocols are $\mathcal{O}(n^2)$ and $\mathcal{O}(n)$ respectively.

The 2-round maxpool protocol invokes the unblind version of the CMP protocol, in which P_2 learns the exact relation between the inputs. We denote the unblind CMP protocol as uCMP$(x, y) :=$ uDReLU$(x - y)$, which simply omits the step 2 and 9 of Alg. 2, i.e., without randomly negating the input and output, while keeping the remaining steps identical.

7. The binary-tree-based Maxpool simply invokes $\mathcal{O}(n)$ times of MAX2 (Eq. 10) for $O(log_2 n)$ rounds.

Based on the uCMP protocol, we can let P_2 compare a list of randomly shuffled inputs which leads to the functionality of determining the maximum element. In particular, P_0 and P_1 first randomly shuffle the n-length ($n > 2$) input vector $\vec{\psi}$ to get $\vec{\phi}$ using seed_{01}. Then, P_0 and P_1 compare each pair of items in $\vec{\phi}$ by invoking the uCMP protocol for $\frac{n \cdot (n-1)}{2}$ times. These invocations form an upper triangular matrix as illustrated in Fig. 5. After computing the output of the comparison matrix $\{\text{uCMP}_{i,j}\}$, P_2 looks for index i^* such that $\forall i < i^\star$, $\text{uCMP}_{i,i^*} = 0$ or NULL and $\forall j > i^\star$, $\text{uCMP}_{i^*,j} = 1$ or NULL. P_2 then constructs a result weight-one indicator vector $\vec{\boldsymbol{\theta}}$ such that $\theta_{i^\star} = 1$, and $\theta_i = 0, \forall i \neq i^\star, i \in [1, n]$. Then, P_2 sends the shares of $\vec{\boldsymbol{\theta}}$ to P_0 and P_1. P_0 and P_1 finally output the shares of the maximum element, i.e., the shares of $\text{MAX}(\vec{\psi}) := \sum_{i=1}^{n} \phi_i \cdot \theta_i$. Notice that the round-collapsing optimization of ReLU could also be applied in this case. We present the details of the algorithm in Alg. 6.

4.6. MIN, SORT, and MED Protocols

Building on the MAX protocol, we could design protocols for the minimum, sorting, and median functionalities. The MIN protocol is similar to the MAX protocol where we only need to change P_2's determination strategy (step 5 of Alg. 6). To determine the minimum element, P_2 looks for the index $i^\star$ subject to $\forall i < i^\star$, $\text{uCMP}_{i,i^\star} = 1$ and $\forall j > i^\star$, $\text{uCMP}_{i^\star,j} = 0$.

For the (descending) SORT protocol, P_2 should continuously look for the maximum element in the matrix. After determining the maximum element (corresponding the indexes $i^\star$), P_2 removes the $i^\star$-th row and the $i^\star$-th column from the matrix. Then, it repeats the above process until the matrix is empty. Fig. 6 illustrates this process (the row and column to be removed are marked in red).

After repeating $n - 1$ times of the above process, P_2 could construct a permutation matrix $\boldsymbol{M}$, in which $m_{i,j} = 1$ if ψ_i is the j-th maximum value and $m_{i,j} = 0$ otherwise. Finally, P_0 and P_1 multiply $\boldsymbol{M}$ by $\vec{\boldsymbol{\phi}}$ in the secret sharing form to acquire the shared sorted arrays.

Since the MAX protocol involves multiplications of the i-th row of the matrix and $\vec{\boldsymbol{\phi}}$ for $\forall i \in [1, n]$, we only need n numbers of triples b to mask $\vec{\boldsymbol{\phi}}$. Also notice that we can apply the balancing technique proposed in Sect. 2 to reduce the amount of one-way communication.

Similarly, in the ascending order SORT protocol, P_2 continuously searches for the minimum element. Note that, P_2 could simultaneously search for the maximum and minimum element in both the descending and ascending order SORT protocols, then eliminate two rows and two columns at the same time, which accelerates the sorting algorithm.

For the MED protocol, we could let P_2 repeat the step 5 in Alg. 6 for $\lceil \frac{n}{2} \rceil$ times. Then, P_2 constructs the indicator vector to reflect the index corresponding to the median element. The remaining steps of MED are the same as those of the MAX protocol.

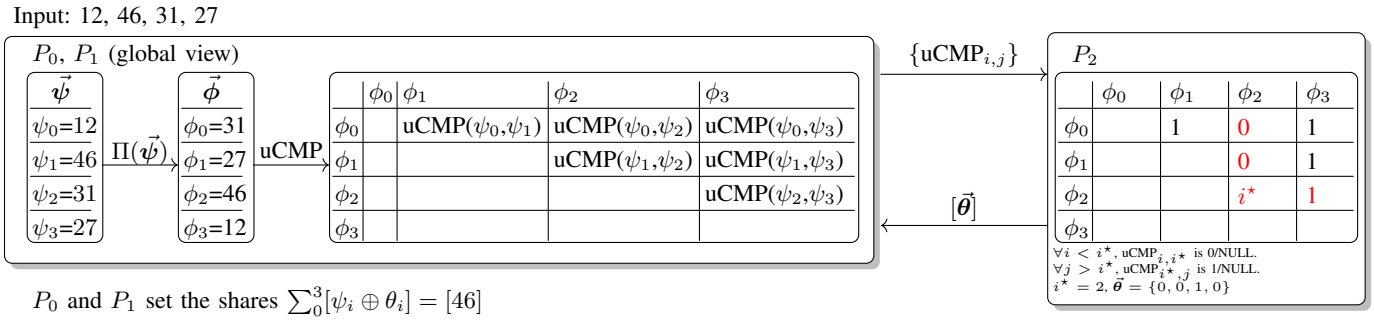


Figure 5: MAX example.

TABLE 7: The communication overhead of our protocols. (DReLU includes DReLU/MSB/BitExt/CMP. ReLU includes ReLU/ABS/Dynamic ReLU/MAX2/MIN2/Funnel ReLU. ℓ: the bit length of an integer ring $\mathbb{Z}_q = [0, q-1]$ ($\ell := \log_2 q$). ℓ_x: the precision of the input. $m+2$ piecewise for PLU. n inputs for MAX/MIN/SORT/MED. Prep.: preprocessing. Rnd.: round. Comm_{ij}: the communication overhead from p_i to p_j, counting in bit. $\textbf{Comm}_{\text{total}}$: the total communication overhead among the three participants. Comm_{02} or Comm_{12} consumes the largest amount of communication in our protocols, which is the dominant one-way communication.)

Protocol	Prep.	Rnd.	Comm_{01}	Comm_{02}	Comm_{12}	Comm_{10}	Comm_{20}	Comm_{21}	$\text{Comm}_{\text{total}}$
DReLU	No	2	N.A.	$(\ell_x+2)\ell$	$(\ell_x+2)\ell$	N.A.	ℓ	ℓ	$(2\ell_x+4)\ell$
ReLU	No	2	ℓ	$(\ell_x+2)\ell$	$(\ell_x+2)\ell$	ℓ	ℓ	2ℓ	$(2\ell_x+9)\ell$
PLU	No	2	ℓ	$(m+1)(\ell_x+2)\ell$	$(m+1)(\ell_x+2)\ell$	ℓ	$\lceil\frac{5m+10}{2}\rceil\ell$	$\lfloor\frac{5m+10}{2}\rfloor\ell$	$((2m+1)\ell_x+9m+16)\ell$
RELU6	No	2	ℓ	$2(\ell_x+2)\ell$	$2(\ell_x+2)\ell$	ℓ	$\lceil\frac{15}{2}\rceil\ell$	$\lfloor\frac{15}{2}\rfloor\ell$	$(4\ell_x+21)\ell$
MAX/MIN	No	2	$n\ell$	$\frac{n(n-1)}{2}(\ell_x+2)\ell$	$\frac{n(n-1)}{2}(\ell_x+2)\ell$	$n\ell$	$\lceil\frac{3n}{2}\rceil\ell$	$\lfloor\frac{3n}{2}\rfloor\ell$	$((2+\ell_x)n^2+(3-\ell_x)n)\ell$
SORT	No	2	$n\ell$	$\frac{n(n-1)}{2}(\ell_x+2)\ell$	$\frac{n(n-1)}{2}(\ell_x+2)\ell$	$n\ell$	$\lceil\frac{3n^2}{2}\rceil\ell$	$\lfloor\frac{3n^2}{2}\rfloor\ell$	$((5+\ell_x)n^2-\ell_x n)\ell$
MED	No	2	$n\ell$	$\frac{n(n-1)}{2}(\ell_x+2)\ell$	$\frac{n(n-1)}{2}(\ell_x+2)\ell$	$n\ell$	$\lceil\frac{3n}{2}\rceil\ell$	$\lfloor\frac{3n}{2}\rfloor\ell$	$((2+\ell_x)n^2+(3-\ell_x)n)\ell$

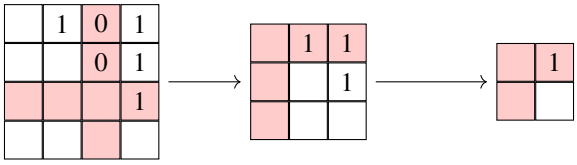


Figure 6: P_2 repeatedly looks for the index of the maximum value and removes the corresponding row and column in the descending order SORT protocol.

5. Evaluation and Experiments

We first evaluate the communication overhead of our proposed protocols in Sect. 5.1, and then run experiments to show the practical performance of our protocols in Sect. 5.2.

5.1. Evaluation

We calculate the theoretical communication overhead in Tab. 7 to have a better view of how the communication overhead dominates the performance of an MPC protocol. Comm_{01}, Comm_{02}, and Comm_{12} denote the uni-direction communication overhead between P_0 and P_1, P_0 and P_2, and P_0 and P_1, respectively, while Comm_{10}, Comm_{20}, and Comm_{21} correspond to the reverse directions. Comm_{total} donates the total communication overhead for each protocol. If more than one triple is used in a protocol, we consider a setting in which P_2 distributes around half of $[c]$'s to P_0, while delivering the other half to P_1. Among our protocols, Comm_{02} or Comm_{12} has the greatest communication overhead comparing with other one-way communication pass, i.e., Comm_{10}, Comm_{01}, etc. Hence, for fairness we use Comm_{02} to compare with that of other related protocols in Tab. 1.

5.2. Experiments

All experiments are carried out under LAN settings, i.e., three cloud virtual machines (VM) in the same region, and we select $q = 2^{64}$ and $\ell_x = 13$ as our system settings. Each VM has 8 CPUs with 2.2GHz (Intel(R) Xeon(R) Gold 6161) and 8GiB memory. The 64Byte-size PING latency between each two VMs is around 250μs to 500μs. The bandwidth (measured by netperf) between each participant ranges from 1250-1780MBytes/s.

We run experiments on our protocols and two state-of-the-art protocols, Falcon [12], [27] and Edabits [25], [28]. The results are shown in Tab. 8 and 9. [8] We also carry out experiments on EQ/ReLU6/MED4/SORT4 and the results are shown in Tab. 10. We provide comparison between our DReLU/ReLU/MAX4/MAX9 and the aforementioned works, and a brief analysis in the following paragraphs.

DReLU and ReLU. When the batch size is 100,000, we achieve more than 390,000 DReLU and 370,000 ReLU

8. We select the honest-majority and passive-secure settings in Edabits and Falcon. Edabits [25] does not have a Maxpool design.

operations per second, respectively. It is worth mentioning that the source codes of Falcon [27] and Edabits [28] do not contain the preprocessing which is heavily computed, and hence, the experiment results of the latency also do not include the preprocessing. By comparing the latency of our protocols (around 340 ms) with the latency for online phase only of Falcon [27] and Edabits [28], our work has a one or two orders of magnitude improvement, respectively, without batching.

Maxpool. Our Maxpool protocols achieve the throughput rate of 110,000 or 41,000 operations per second, for binary-tree-based MAX4 or MAX9, respectively. When the bandwidth is not saturated, i.e., small batch size and small n, both our 2-round and binary-tree-based Maxpool protocols outperform Falcon's protocols. When the batch size is large, our binary-tree-based Maxpool protocols still have better performance than Falcon's. Once again, the latency recorded from the Falcon protocols does not contain its heavy preprocessing.

Estimated E2E inference. Based on the above experiment results, we could give an estimation of the end-to-end improvement in PPML by relating the obtained speed-ups on ReLU back with Fig. 1. We use Cifar10_vgg16_32 as an example, such a model contains 16 ReLU layers each with batch size ranging from 32×100 to $32\times65{,}535$. Using our experiment results, we calculate the total time needed for operating all ReLU layers using our protocols is around 15.3 s. By substituting this time for the time used in Figure 1, i.e., 106 s (time for all ReLU operations) out of 126 s (time for the whole inference), we could achieve a $3.6\times$ of improvement on the complete interface. However, this estimation omits the difference between CPU and GPU platforms. We are working on the GPU integration of Bicoptor protocols and will provide a more rigorous experiment result on the end-to-end improvement in future work.

6. Conclusion

We propose Bicoptor, which includes several two-round three-party computation protocols without preprocessing for PPML. We innovate a novel sign determination protocol, which relies on a clever use of the truncation protocol. Bicoptor supports the DReLU, Equality, ABS, ReLU, Dynamic ReLU (Leaky ReLU, PReLU, RReLU), ReLU6, Piecewise Linear Unit (PLU), MAX, MIN, SORT, and median (MED) functions. The experiments exhibit our fast performance, which has a one or two orders of magnitude improvement to ReLU in the state-of-the-art works, i.e., Falcon or Edabits, respectively.

Acknowledgment

We would like to thank Zhongkai Li, Wenyuan Tian, Xianggui Wang, and Hao Guo for their great help in the implementations of Bicoptor. Yu Yu was supported by the National Natural Science Foundation of China (Grant Nos.62125204, 92270201 and 61872236). Yu Yu also acknowledges the support from the XPLORER PRIZE.

References

[1] A. C. Yao, "Protocols for secure computations (extended abstract)," in FOCS 1982, pp. 160–164.

[2] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game or A completeness theorem for protocols with honest majority," in STOC 1987, 1987, pp. 218–229.

[3] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract)," in STOC 1988, pp. 1–10.

[4] D. Beaver, "Efficient multiparty protocols using circuit randomization," in CRYPTO 1991, pp. 420–432.

[5] D. Chaum, C. Crépeau, and I. Damgård, "Multiparty unconditionally secure protocols (extended abstract)," in STOC 1988, pp. 11–19.

[6] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "GAZELLE: A low latency framework for secure neural network inference," in USENIX Security 2018, pp. 1651–1669.

[7] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference service for neural networks," in USENIX Security 2020, pp. 2505–2522.

[8] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow2: Practical 2-party secure inference," in CCS 2020. ACM, pp. 325–342.

[9] A. Patra, T. Schneider, A. Suresh, and H. Yalame, "ABY2.0: Improved Mixed-Protocol secure Two-Party computation," in USENIX Security 21, pp. 2165–2182.

[10] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, "Chameleon: A hybrid secure computation framework for machine learning applications," in AsiaCCS 2018, pp. 707–721.

[11] S. Wagh, D. Gupta, and N. Chandran, "SecureNN: 3-party secure computation for neural network training," Proc. Priv. Enhancing Technol., vol. 2019, no. 3, pp. 26–49, 2019.

[12] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin, "Falcon: Honest-majority maliciously secure framework for private deep learning," Proc. Priv. Enhancing Technol., vol. 2021, no. 1, pp. 188–208, 2021.

[13] S. Tan, B. Knott, Y. Tian, and D. J. Wu, "Cryptgpu: Fast privacy-preserving machine learning on the GPU," in S&P 2021, pp. 1021–1038.

[14] P. Mohassel and P. Rindal, "Aby^3: A mixed protocol framework for machine learning," in CCS 2018, pp. 35–52.

[15] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh, "ASTRA: high throughput 3pc over rings with application to secure prediction," in CCS Workshop 2019, pp. 81–92.

[16] A. Patra and A. Suresh, "BLAZE: blazing fast privacy-preserving machine learning," in NDSS 2020.

[17] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow: Secure tensorflow inference," in S&P 2020, pp. 336–353.

[18] A. P. K. Dalskov, D. Escudero, and M. Keller, "Fantastic four: Honest-majority four-party secure computation with malicious security," in USENIX Security 2021, pp. 2183–2200.

[19] N. Koti, M. Pancholi, A. Patra, and A. Suresh, "SWIFT: super-fast and robust privacy-preserving machine learning," in USENIX 2021, pp. 2651–2668.

[20] M. Byali, H. Chaudhari, A. Patra, and A. Suresh, "FLASH: fast and robust framework for privacy-preserving machine learning," Proc. Priv. Enhancing Technol., vol. 2020, no. 2, pp. 459–480, 2020.

[21] H. Chaudhari, R. Rachuri, and A. Suresh, "Trident: Efficient 4pc framework for privacy preserving machine learning," in NDSS 2020.

TABLE 8: The performance of Bicoptor, Falcon [12] and Edabits [25] DReLU and ReLU protocols. (Lat.: latency. ALat.: amortized latency. Thru.: throughput rate. $q = 2^{64},\ \ell_x = 13$.)

Batch Size	Protocol	DReLU			ReLU		
		Lat.(μs)	ALat.(μs)	Thru.(ops/s)	Lat.(us)	ALat.(us)	Thru.(ops/s)
1	Bicoptor	330.2	330.2	3028.6	340.0	340.0	2941.1
	Falcon [12]	1005.8	1005.8	994.3	1184.4	1184.4	844.3
	Edabits [25]	22104.7	22104.7	45.2	24137.9	24137.9	41.4
10^2	Bicoptor	703.0	7.0	142243.3	705.4	7.1	141760.4
	Falcon [12]	1220.0	12.2	81969.5	1374.6	13.7	72748.4
	Edabits [25]	179186.0	1791.9	558.1	180822.0	1808.2	553.0
10^3	Bicoptor	3078.1	3.1	324872.2	3428.9	3.4	291635.4
	Falcon [12]	3515.7	3.5	284435.7	4118.5	4.1	242806.8
	Edabits [25]	486465.7	486.5	2055.6	566871.0	566.9	1764.1
10^4	Bicoptor	25969.9	2.6	385061.2	31531.4	3.2	317144.5
	Falcon [12]	22266.5	2.2	449105.2	33749.6	3.4	296299.8
	Edabits [25]	744525.3	74.5	13431.4	795877.7	79.6	12564.7
10^5	Bicoptor	253397.3	2.5	394637.1	267570.0	2.7	373734.0
	Falcon [12]	322137.0	3.2	310426.9	333221.0	3.3	300101.1
	Edabits [25]	2368510.0	23.7	42220.6	2563846.7	25.6	39003.9

TABLE 9: The performance of binary-tree and 2-round Bicoptor MAX4/MAX9, and Falcon [12] MAX4/MAX9 protocols. (Lat.: latency. ALat.: amortized latency. Thru.: throughput rate. $q = 2^{64},\ \ell_x = 13$. Bicoptor-Binary: the binary-tree-based maxpool Bicoptor protocols.)

Batch Size	Protocol	MAX4			MAX9		
		Lat.(μs)	ALat.(μs)	Thru.(ops/s)	Lat.(us)	ALat.(us)	Thru.(ops/s)
1	Bicoptor-Binary	651.8	651.8	1534.2	1617.0	1617.0	618.4
	Bicoptor-2-round	341.6	341.6	2927.3	455.1	455.1	2197.5
	Falcon [12]	3703.7	3703.7	270.0	10449.5	10449.5	95.7
10^2	Bicoptor-Binary	1718.1	17.2	58202.3	4211.1	42.1	23747.0
	Bicoptor-2-round	2179.3	21.8	45886.4	9673.2	96.7	10337.8
	Falcon [12]	4509.4	45.1	22176.1	14175.3	141.8	7054.5
10^3	Bicoptor-Binary	9783.3	9.8	102214.6	26420.8	26.4	37849.0
	Bicoptor-2-round	16399.8	16.4	60976.5	90154.9	90.2	11092.0
	Falcon [12]	12635.9	12.6	79139.4	34885.1	34.9	28665.6
10^4	Bicoptor-Binary	87470.8	8.7	114323.8	244932.0	24.5	40827.7
	Bicoptor-2-round	165940.2	16.6	60262.7	1015628.1	101.6	9846.1
	Falcon [12]	111099.7	11.1	90009.3	297128.0	29.7	33655.5
10^5	Bicoptor-Binary	858400.8	8.6	116495.7	2404498.0	24.0	41588.7
	Bicoptor-2-round	1723884.6	17.2	58008.5	10410762.7	104.1	9605.4
	Falcon [12]	1063411.3	10.6	94037.0	2802530.0	28.0	35682.0

[22] "Cryptgpu source code," Github, 2021, https://github.com/jeffreysijuntan/CryptGPU.

[23] D. Demmler, T. Schneider, and M. Zohner, "ABY - A framework for efficient mixed-protocol secure two-party computation," in NDSS 2015.

[24] P. Mohassel and Y. Zhang, "SecureML: A system for scalable privacy-preserving machine learning," in S&P 2017, pp. 19–38.

[25] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl, "Improved primitives for mpc over mixed arithmetic-binary circuits," in CRYPTO 2020, pp. 823–852.

[26] X. Wang, A. J. Malozemoff, and J. Katz, "EMP-toolkit: Efficient MultiParty computation toolkit," https://github.com/emp-toolkit, 2016.

[27] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin, "FALCON source code," https://github.com/snwagh/falcon-public, 2021.

[28] M. Keller, "MP-SPDZ, Edabits source code," https://github.com/data61/MP-SPDZ, 2021.

TABLE 10: The performance of Bicoptor EQ, ReLU6, MED4, and SORT6 protocols. (Lat.: latency. ALat.: amortized latency. Thru.: throughput rate. $q = 2^{64},\ \ell_x = 13$.)

Batch	**EQ**			**ReLU6**		
Size	Lat.(μs)	ALat.(μs)	Thru.(ops/s)	Lat.(us)	ALat.(us)	Thru.(ops/s)
1	334.4	334.4	2990.1	355.0	355.0	2817.1
10^2	936.8	9.4	106743.3	1104.9	11.0	90505.5
10^3	5931.9	5.9	168578.9	6685.8	6.7	149571.5
10^4	63747.3	6.4	156869.3	62503.6	6.3	159990.8
10^5	528272.7	5.3	189296.2	620112.0	6.2	161261.2
Batch	**MED4**			**SORT4**		
Size	Lat.(μs)	ALat.(μs)	Thru.(ops/s)	Lat.(us)	ALat.(us)	Thru.(ops/s)
1	361.9	361.9	2763.3	381.6	381.6	2620.5
10^2	2282.3	22.8	43815.8	2506.5	25.1	39895.7
10^3	17488.1	17.5	57181.7	21285.4	21.3	46980.6
10^4	169422.3	16.9	59024.1	193220.6	19.3	51754.3
10^5	1906699.0	19.1	52446.7	2425505.4	24.3	41228.5

TABLE 11: An example of the fixed-point computation for a decimal multiplication ($q = 2^{40},\ \ell_x = 16$).

Decimal	**Binary**
α=10.82421875	00000000_00000000_00000000_00001010_11010011
β=6.2265625	00000000_00000000_00000000_00000110_00111010
γ=−6.2265625	11111111_11111111_11111111_11111011_11000110
TRC($\alpha \cdot \beta$,8) =67.39453125	00000000_00000000_00000000_01000011_01100101
TRC($\alpha \cdot \gamma$,8) =−67.39453125	11111111_11111111_11111111_11101110_00100010

Appendix A.
Decimal Arithmetic with Fixed-point Computation

In a practical PPML application, the inputs and coefficients could be decimal numbers. To express a decimal number, we consider a typical case in which the bit length of an integer is the same as that of the fractional part, i.e., the last $\frac{\ell_x}{2}$ bits are used to represent the fractional part. Hence, $x := \sum_{i=0}^{\ell_x - 1} x_i \cdot 2^{i - \frac{\ell_x}{2}}$ where x_i is the i-th bit of x. This expression works for both positive and negative decimal numbers. After a multiplication, a truncation is required to recover the original precision. Tab. 11 shows an example.

Appendix B.
Non-linear Function Definition in PPML

Bit-wise univariate function. From DReLU, the bit extraction (BitExt) and the most significant bit (MSB) functions are obtained as Eq. 3.

$$\begin{aligned} &\text{BitExt}(x) = \text{MSB}(x) = \begin{cases} 0, & x \geq 0 \\ 1, & x < 0 \end{cases} \\ =&1 - \text{DReLU}(x) \end{aligned} \tag{3}$$

Univariate function with two segments. Other functions could also be derived from DReLU, such as ABS (Eq. 4), ReLU (Eq. 1), and Dynamic ReLU (Eq. 5). [9]

$$\begin{aligned} &\text{ABS}(x) = \begin{cases} x, & x \geq 0 \\ -x, & x < 0 \end{cases} \\ =&\text{DReLU}(x) \cdot x + (\text{DReLU}(x) - 1) \cdot x \\ =&(2 \cdot \text{DReLU}(x) - 1) \cdot x \end{aligned} \tag{4}$$

$$\begin{aligned} &\text{Dynamic ReLU}(x) = \begin{cases} \alpha_1 \cdot x & x \geq 0 \\ \alpha_0 \cdot x & x < 0 \end{cases} \\ &(\alpha_0, \alpha_1 \text{ are two constants.}) \\ =&\alpha_1 \cdot \text{DReLU}(x) \cdot x + \alpha_0 \cdot (1 - \text{DReLU}(x)) \cdot x \\ =&(\alpha_0 + (\alpha_1 - \alpha_0) \cdot \text{DReLU}(x)) \cdot x \end{aligned} \tag{5}$$

Comparison function. From DReLU, we can also formulate the comparison function(CMP) and the Equality function (Eq. 7).

$$\text{CMP}(x_0, x_1) = \begin{cases} 1 & x_0 \geq x_1 \\ 0 & x_0 < x_1 \end{cases} = \text{DReLU}(x_0 - x_1) \tag{6}$$

$$\begin{aligned} &\text{Equality}(x_0, x_1) = \begin{cases} 1 & x_0 = x_1 \\ 0 & x_0 \neq x_1 \end{cases} \\ =&1 - \text{CMP}(x_0, x_1) \oplus \text{CMP}(x_1, x_0) \\ =&1 - \text{DReLU}(x_0 - x_1) \oplus \text{DReLU}(x_1 - x_0) \end{aligned} \tag{7}$$

9. Leaky ReLU, PReLU, and RReLU are the special cases of Dynamic ReLU. For Leaky ReLU, $\alpha_0 = 0.001, \alpha_1 = 1$. For PReLU, α_0 is a pretrained constant and $\alpha_1 = 1$. For RReLU, α_0 is a random constant and $\alpha_1 = 1$.

The piecewise linear unit (PLU, Eq. 8) function has more than two input intervals, utilizing the CMP function.

$$\begin{aligned}
\text{PLU}(x) = &\begin{cases}
\alpha_{m+1}\cdot x+\beta_{m+1}, & \gamma_m <= x\\
\alpha_m\cdot x+\beta_m, & \gamma_{m-1} <= x < \gamma_m\\
... & \\
\alpha_j\cdot x+\beta_j, & \gamma_{j-1} <= x < \gamma_j\\
... & \\
\alpha_1\cdot x+\beta_1, & \gamma_0 <= x < \gamma_1\\
\alpha_0\cdot x+\beta_0, & x < \gamma_0
\end{cases}\\
&(m+2 \text{ piecewise. } \alpha_j \text{ and } \beta_j\ (\forall j\in[0,m+1]),\\
&\text{and } \gamma_j\ (\forall j\in[0,m]) \text{ are constants.})\\
=&(\text{CMP}(x,\gamma_m)\oplus 0)\cdot(\alpha_{m+1}\cdot x+\beta_{m+1})+...\\
&+(\text{CMP}(x,\gamma_{j-1})\oplus\text{CMP}(x,\gamma_j))\cdot(\alpha_j\cdot x+\beta_j)\\
&+...+(1\oplus\text{CMP}(x,\gamma_0))\cdot(\alpha_0\cdot x+\beta_0)\\
=&(\text{DReLu}(x-\gamma_m)\oplus 0)\cdot(\alpha_{m+1}\cdot x+\beta_{m+1})+...\\
&+(\text{DReLu}(x-\gamma_{j-1})\oplus\text{DReLu}(x-\gamma_j))\cdot(\alpha_j\cdot x+\beta_j)\\
&+...+(1\oplus\text{DReLu}(x-\gamma_0))\cdot(\alpha_0\cdot x+\beta_0)
\end{aligned} \tag{8}$$

The ReLU6 (Eq. 9) function is a special case of PLU with three input intervals, i.e., $m=1,\alpha_0=\beta_0=\beta_1=\alpha_2=\gamma_0=0,\alpha_1=1,\beta_2=\gamma_2=6$.

$$\begin{aligned}
\text{ReLU6}(x) = &\begin{cases}
6 & 6 <= x\\
x & 0 <= x < 6\\
0 & x < 0
\end{cases}\\
=&(\text{DReLu}(x-6)\oplus 0)\cdot 6\\
&+(\text{DReLu}(x-0)\oplus\text{DReLu}(x-6))\cdot x\\
&+(1\oplus\text{DReLu}(x))\cdot 0
\end{aligned} \tag{9}$$

The CMP can also derive the maximum (MAX2, Eq. 10) and minimum (MIN2, Eq. 11) functions for two inputs.

$$\begin{aligned}
\text{MAX2}(x,y) = &\begin{cases}
x & x\geq y\\
y & x < y
\end{cases}\\
=&\text{CMP}(x,y)\cdot x+(1-\text{CMP}(x,y))\cdot y\\
=&\text{DReLU}(x-y)\cdot x+(1-\text{DReLU}(x-y))\cdot y\\
=&\text{DReLU}(x-y)\cdot(x-y)+y=\text{ReLU}(x-y)+y
\end{aligned} \tag{10}$$

$$\begin{aligned}
\text{MIN2}(x,y) = &\begin{cases}
y & x\geq y\\
x & x < y
\end{cases}\\
=&\text{CMP}(x,y)\cdot y+(1-\text{CMP}(x,y))\cdot x\\
=&x-\text{ReLU}(x-y)=\text{ReLU}(y-x)+x
\end{aligned} \tag{11}$$

Funnel ReLU (Eq. 12) is extended from MAX2.

$$\begin{aligned}
&\text{Funnel ReLU}(x)=\text{MAX2}(x,\text{T}(x))\\
&(\text{T}(x)\text{is a linear function.})\\
=&\text{ReLU}(x-\text{T}(x))+\text{T}(x)
\end{aligned} \tag{12}$$

When the number of inputs is more than two, we could compute the MAX, MIN, sorting (SORT), and median (MED) functions. The typical Maxpool function in machine learning is to determine the maximum element from several inputs, e.g., four or nine.

Appendix C. Proofs

We prove Theorem 1 by the following lemmas. We first prove Lemma 1 which is stated in [24, Sect. 4.1] but not proved. Lemma 1 explains the one-bit error phenomenon, where Lemma 2 presents a special case of Lemma 1, i.e., when errors do occur. Lemma 3 further proves the inevitable existence 1 or $q-1$, which implies Lemma 4. Lemma 4 exhibits the maximum truncation length required to output 1 or $q-1$. Lemma 5 and Lemma 6 prove the pattern of the tail elements in the truncation result array. Finally, the proofs of these lemmas imply Theorem 1, i.e., the pattern of the truncation result array.

In the following, we define $\xi=\xi(x):=x$ if the input $x\in[0,2^{\ell_x})$ and $\xi=\xi(x):=q-x$ if $x\in(q-2^{\ell_x},q)$, i.e., ξ stays in $[0,2^{\ell_x})$. For an x in $\mathbb{Z}_q=[0,q-1]$, the bit-length of q and x is ℓ and ℓ_x, respectively ($\ell>\ell_x$). The binary form of ξ is defined as $\{\xi_{\ell_x-1},\xi_{\ell_x-2},\cdots,\xi_1,\xi_0\}$, in which ξ_i denotes the i-th bit and $\xi:=\sum_{i=0}^{\ell_x-1}\xi_i\cdot 2^i$. λ is the effective bit length of ξ, i.e., $\xi_{\lambda-1}=1$ and $\lambda+1<\ell$. The shares of x are $[x]_0$ and $[x]_1$ for P_0 and P_1, respectively, in which $[x]_0=r$, $[x]_1=x-[x]_0$ and r is a random number from $\mathbb{Z}_q$. Moreover, we define

$$\begin{aligned}
&\frac{[x]_0}{2^k}:=\text{rShift}([x]_0,k)=[\text{TRC}(x,k)]_0,\\
&\frac{q-[x]_0}{2^k}:=\text{rShift}(q-[x]_1,k),\\
&[\text{TRC}(x,k)]_1=q-\text{rShift}(q-[x]_1,k),\\
&\\
&r:=r''\cdot 2^k+r',\ r''\in[0,2^{\ell-k}),\ r'\in[0,2^k),\\
&\xi:=\xi''\cdot 2^k+\xi',\ \xi''\in[0,2^{\ell-k}),\ \xi'\in[0,2^k),
\end{aligned}$$

then $\frac{\xi}{2^k}=\xi''$, for Lemma 1 and Lemma 2.

The proof of Lemma 1:

Proof: **Case I**: If $x\in[0,2^{\ell_x})$, then $\xi=x$. Since $[x]_0=r$, $[x]_1=x-[x]_0$ and $r\in\mathbb{Z}_q$, the probability of $r\geq\xi$ is $\frac{q-\xi}{q}$, which is larger than $\frac{q-2^{\ell_x}}{q}=1-\frac{2^{\ell_x}}{q}>1-\frac{2^{\ell_x}}{2^{\ell-1}}=1-\frac{1}{2^{\ell-\ell_\xi-1}}=1-2^{\ell_x+1-\ell}$. Hence, we will discuss this case under the condition $r\geq\xi$. Let $\mathsf{ret}:=r-\xi=\mathsf{ret}''\cdot 2^k+\mathsf{ret}'$, where $\mathsf{ret}''\in[0,2^{\ell-k})$ and $\mathsf{ret}'\in[0,2^k)$. Then, $\frac{\mathsf{ret}}{2^k}=\mathsf{ret}''$. Moreover, $r-\xi=r''\cdot 2^k+r'-\xi''\cdot 2^k-\xi'=r''\cdot 2^k-\xi''\cdot 2^k+r'-\xi'=(r''-\xi'')\cdot 2^k+(r'-\xi')$. Next, we will prove that $\mathsf{ret}''=r''-\xi''-\mathsf{bit}$, where $\mathsf{bit}=0$ or 1.

- For $r'\geq\xi'$: Due to $r'\in[0,2^k)$ and $\xi'\in[0,2^k)$, $r'-\xi'\in[0,2^k)$. Since $r\geq\xi$, $r''\geq\xi''$. Then, $r-\xi=r''\cdot 2^k-\xi''\cdot 2^k+r'-\xi'=(r''-\xi'')\cdot 2^k+(r'-\xi')$ satisfying $r''-\xi''\in[0,2^{\ell-k})$ and $r'-\xi'\in[0,2^k)$. Therefore, $\mathsf{ret}''=(r-\xi)/2^k=r''-\xi''$.
- For $r'<\xi'$: Due to $r'\in[0,2^k)$ and $\xi'\in[0,2^k)$, $2^k+r'-\xi'\in[0,2^k)$. Since $r\geq\xi$, $r''\geq\xi''+1$. Then,

$r-\xi = r''\cdot 2^k - \xi''\cdot 2^k + r' - \xi' = (r''-\xi''-1)\cdot 2^k + (2^k + r' - \xi')$ satisfying $r''-\xi''-1 \in [0, 2^{\ell-k})$ and $2^k + r' - \xi' \in [0, 2^k)$. Therefore, $\mathsf{ret}'' = (r-\xi)/2^k = r''-\xi''-1$.

Hence, $\frac{r-\xi}{2^k} = \mathsf{ret}'' = r''-\xi''-\mathsf{bit}$, where $\mathsf{bit}=0$ or 1, with a probability larger than $1-2^{\ell_x+1-\ell}$. Since $[\mathrm{TRC}(x,k)]_0 = \frac{[x]_0}{2^k} = \frac{r}{2^k} = r''$ and $[\mathrm{TRC}(x,k)]_1 = q - \frac{q-[x]_1}{2^k} = q - \frac{r-\xi}{2^k} = q-(r''-\xi''-\mathsf{bit})$, $\mathrm{TRC}(x,k) = [\mathrm{TRC}(x,k)]_0 + [\mathrm{TRC}(x,k)]_1 \bmod q = r'' + q - (r''-\xi''-\mathsf{bit}) \bmod q = q+\xi''+\mathsf{bit} \bmod q = \xi''+\mathsf{bit} \bmod q = \frac{\xi}{2^k} + \mathsf{bit}, \mathsf{bit}=0$ or 1 with a probability larger than $1-2^{\ell_x+1-\ell}$.

Case II: If $x \in (q-2^{\ell_x}, q)$, then $\xi = q-x$. Since $[x]_0 = r$, $[x]_1 = x-[x]_0$ and $r \in \mathbb{Z}_q$, the probability of $\xi + r < q$ is $\frac{q-\xi}{q}$, which is larger than $\frac{q-2^{\ell_x}}{q} = 1-\frac{2^{\ell_x}}{q} > 1-\frac{2^{\ell_x}}{2^{\ell-1}} = 1-\frac{1}{2^{\ell-\ell_\xi-1}} = 1-2^{\ell_x+1-\ell}$. Hence, we will discuss this case under $\xi + r < q$. Let $\mathsf{ret} := r+\xi = \mathsf{ret}''\cdot 2^k + \mathsf{ret}'$, where $\mathsf{ret}'' \in [0, 2^{\ell-k})$ and $\mathsf{ret}' \in [0, 2^k)$. Then, $\frac{\mathsf{ret}}{2^k} = \mathsf{ret}''$. Moreover, $r+\xi = r''\cdot 2^k + r' + \xi''\cdot 2^k + \xi' = r''\cdot 2^k + \xi''\cdot 2^k + r' + \xi' = (r''+\xi'')\cdot 2^k + (r'+\xi')$. Next, we will prove that $\mathsf{ret}'' = r''+\xi''+\mathsf{bit}$, where $\mathsf{bit}=0$ or 1.

- For $r'+\xi' < 2^k$: Since $\xi + r < q$, $r+\xi = (r''+\xi'')\cdot 2^k + (r'+\xi')$, and $r'+\xi' \in [0, 2^k)$, we have $r''+\xi'' \in [0, 2^{\ell-k})$. Then, $r+\xi = r''\cdot 2^k + \xi''\cdot 2^k + r' + \xi' = (r''+\xi'')\cdot 2^k + (r'+\xi')$ satisfying $r''+\xi'' \in [0, 2^{\ell-k})$ and $r'+\xi' \in [0, 2^k)$. Therefore, $\mathsf{ret}'' = (r+\xi)/2^k = r''+\xi''$.
- For $r'+\xi' \geq 2^k$: Due to $r' \in [0, 2^k)$ and $\xi' \in [0, 2^k)$, $r'+\xi'-2^k \in [0, 2^k)$. Since $\xi + r < q$ and $r+\xi = (r''+\xi'')\cdot 2^k + (r'+\xi') = (r''+\xi''+1)\cdot 2^k + (r'+\xi'-2^k)$, we have $r''+\xi''+1 \in [0, 2^{\ell-k})$. Then, $r+\xi = (r''+\xi''+1)\cdot 2^k + (r'+\xi'-2^k)$ satisfying $r''+\xi''+1 \in [0, 2^{\ell-k})$ and $r'+\xi'-2^k \in [0, 2^k)$. Therefore, $\mathsf{ret}'' = (r+\xi)/2^k = r''+\xi''+1$.

Hence, $\frac{r+\xi}{2^k} = \mathsf{ret}'' = r''+\xi''+\mathsf{bit}$, where $\mathsf{bit}=0$ or 1, with a probability larger than $1-2^{\ell_x+1-\ell}$. Since $[\mathrm{TRC}(x,k)]_0 = \frac{[x]_0}{2^k} = \frac{r}{2^k} = r''$ and $[\mathrm{TRC}(x,k)]_1 = q - \frac{q-[x]_1}{2^k} = q - \frac{q-(x-r)}{2^k} = q - \frac{q-(q-\xi-r)}{2^k} = q - \frac{\xi+r}{2^k} = q-(r''-\xi''-\mathsf{bit})$, then $\mathrm{TRC}(x,k) = [\mathrm{TRC}(x,k)]_0 + [\mathrm{TRC}(x,k)]_1 \bmod q = r''+q-(r''+\xi''+\mathsf{bit}) \bmod q = q-\xi''-\mathsf{bit} = q-\frac{\xi}{2^k} - \mathsf{bit}, \mathsf{bit}=0$ or 1 with a probability larger than $1-2^{\ell_x+1-\ell}$. □

Based on Lemma 1, Lemma 2 describes when the one-bit error would exist.

Lemma 2. If $\ell := \log_2 q > \ell_x + 1$, $[x]_0 = r$, and $[x]_1 = x - [x]_0 \bmod q$, the following results hold with a probability of $1-2^{\ell_x+1-\ell}$:

- If $x = \xi$, then $\mathrm{TRC}(x,k) = \frac{\xi}{2^k}$ for $r' \geq \xi'$, and $\mathrm{TRC}(x,k) = \frac{\xi}{2^k}+1$ for $r' < \xi'$.
- If $x = q-\xi$, then $\mathrm{TRC}(x,k) = q - \frac{\xi}{2^k}$ for $r'+\xi' < 2^k$, and $\mathrm{TRC}(x,k) = q - \frac{\xi}{2^k} - 1$ for $r'+\xi' \geq 2^k$.

Lemma 2 can be proved in the similar way as Lemma 1.

We define the following notations $r''_\lambda, r'_\lambda, r''_{\lambda-1}, r'_{\lambda-1}$, $\xi''_\lambda, \xi'_\lambda, \xi''_{\lambda-1}, \xi'_{\lambda-1}$ for Lemma 3 and Lemma 4.

$$
\begin{aligned}
r :=& r''_\lambda \cdot 2^\lambda + r'_\lambda,\ r''_\lambda \in [0, 2^{\ell-\lambda}),\ r'_\lambda \in [0, 2^\lambda),\\
r :=& r''_{\lambda-1} \cdot 2^{\lambda-1} + r'_{\lambda-1},\ r''_{\lambda-1} \in [0, 2^{l-\lambda+1}),\\
& r' \in [0, 2^{\lambda-1}),\\
\xi :=& \xi''_\lambda \cdot 2^\lambda + \xi'_\lambda,\ \xi''_\lambda \in [0, 2^{\ell-\lambda}),\ \xi'_\lambda \in [0, 2^\lambda),\\
\xi :=& \xi''_{\lambda-1} \cdot 2^{\lambda-1} + \xi'_{\lambda-1},\ \xi''_{\lambda-1} \in [0, 2^{\ell-\lambda+1}),\\
& \xi'_{\lambda-1} \in [0, 2^{\lambda-1}).
\end{aligned}
$$

Lemma 3. Let λ be the effective bit length of ξ, i.e., $\xi_{\lambda-1} = 1$. Let η be the λ-th significant bit of r, i.e., $\eta := r_{\lambda-1}$. If $\lambda+1 < \ell$, the following results hold with a probability of $1-2^{\lambda+1-\ell}$:

- If $x = \xi$, then $\mathrm{TRC}(x, \lambda-1) = 1$ or $\mathrm{TRC}(x,\lambda) = 1$.
- If $x = q-\xi$, then $\mathrm{TRC}(x,\lambda-1) = q-1$ or $\mathrm{TRC}(x,\lambda) = q-1$.

Proof: **Case I**: For $x = \xi$, Lemma 1 implies that $\mathrm{TRC}(\xi, \lambda-1) = \frac{\xi}{2^{\lambda-1}} + \mathsf{bit} = \xi_{\lambda-1} + \mathsf{bit} = 1 + \mathsf{bit}$, $\mathsf{bit}=0$ or 1 with a probability of $1-2^{\lambda+1-\ell}$. If $\mathsf{bit}=0$, then $\mathrm{TRC}(\xi, \lambda-1) = 1$. If $\mathsf{bit}=1$, Lemma 2 implies $r'_{\lambda-1} < \xi'_{\lambda-1}$. Since $\xi_{\lambda-1} = 1$, $r'_\lambda = \eta\cdot 2^{\lambda-1} + r'_{\lambda-1}$ and $\xi'_\lambda = \xi_{\lambda-1}\cdot 2^{\lambda-1} + \xi'_{\lambda-1} = 2^{\lambda-1} + \xi'_{\lambda-1}$. Then, $r'_\lambda < x'_\lambda$. Lemma 2 implies $\mathrm{TRC}(\xi, \lambda) = \frac{\xi}{2^\lambda} + 1 = 0 + 1 = 1$.

Case II: For $x = q-\xi$, Lemma 1 implies that $\mathrm{TRC}(q-\xi, \lambda-1) = q - \frac{\xi}{2^{\lambda-1}} - \mathsf{bit} = q - \xi_{\lambda-1} - \mathsf{bit} = q-1-\mathsf{bit}$, $\mathsf{bit}=0$ or 1 with a probability of $1-2^{\lambda+1-\ell}$. If $\mathsf{bit}=0$, $\mathrm{TRC}(q-\xi, \lambda-1) = q-1$. If $\mathsf{bit}=1$, Lemma 2 implies $r'_{\lambda-1} + \xi'_{\lambda-1} \geq 2^{\lambda-1}$. Therefore, $r'_\lambda = \eta\cdot 2^{\lambda-1} + r'_{\lambda-1}$ and $\xi'_\lambda = \xi_{\lambda-1}\cdot 2^{\lambda-1} + \xi'_{\lambda-1} = 2^{\lambda-1} + \xi'_{\lambda-1}$. Then, $r'_\lambda + \xi'_\lambda = r'_{\lambda-1} + \xi'_{\lambda-1} + 2^{\lambda-1} \geq 2^\lambda$ due to $r'_{\lambda-1} + \xi'_{\lambda-1} \geq 2^{\lambda-1}$. Lemma 2 implies $\mathrm{TRC}(q-\xi, \lambda) = q - \frac{\xi}{2^\lambda} - 1 = q-0-1 = q-1$. □

Lemma 4. If $\lambda+1 < \ell$, for any number $\hat{\ell} \geq \lambda$, the following results hold with a probability of $1-2^{\lambda+1-\ell}$:

- There exists a number $\hat{\lambda} \leq \hat{\ell}$, satisfying $\mathrm{TRC}(\xi, \hat{\lambda}) = 1$.
- There exists a number $\hat{\lambda} \leq \hat{\ell}$, satisfying $\mathrm{TRC}(\xi, \hat{\lambda}) = q-1$.

Lemma 4 could be proved via the proof of Lemma 3.

Considering any number $\lambda' > \ell_x$, we extra define the following notations $r''_{\lambda'}, r'_{\lambda'}, r''_{\lambda'-1}, r'_{\lambda'-1}, \xi''_{\lambda'}, \xi'_{\lambda'}, \xi''_{\lambda'-1}$, $\xi'_{\lambda'-1}$ for Lemma 5 and Lemma 6.

$$
\begin{aligned}
r :=& r''_{\lambda'} \cdot 2^{\lambda'} + r'_{\lambda'},\ r''_{\lambda'} \in [0, 2^{\ell-\lambda'}),\ r'_{\lambda'} \in [0, 2^{\lambda'}),\\
r :=& r''_{\lambda'-1} \cdot 2^{\lambda'-1} + r'_{\lambda'-1},\ r''_{\lambda'-1} \in [0, 2^{l-\lambda'+1}),\\
& r' \in [0, 2^{\lambda'-1}),\\
\xi :=& \xi''_{\lambda'} \cdot 2^{\lambda'} + \xi'_{\lambda'},\ \xi''_{\lambda'} \in [0, 2^{\ell-\lambda'}),\ \xi'_{\lambda'} \in [0, 2^{\lambda'}),\\
\xi :=& \xi''_{\lambda'-1} \cdot 2^{\lambda'-1} + \xi'_{\lambda'-1},\ \xi''_{\lambda'-1} \in [0, 2^{\ell-\lambda'+1}),\\
& \xi'_{\lambda'-1} \in [0, 2^{\lambda'-1}).
\end{aligned}
$$

Lemma 5. If $\ell > \lambda+1$, for any number $\lambda' > \ell_x$, the following results hold with a probability of $1-2^{\lambda'-\ell}$:

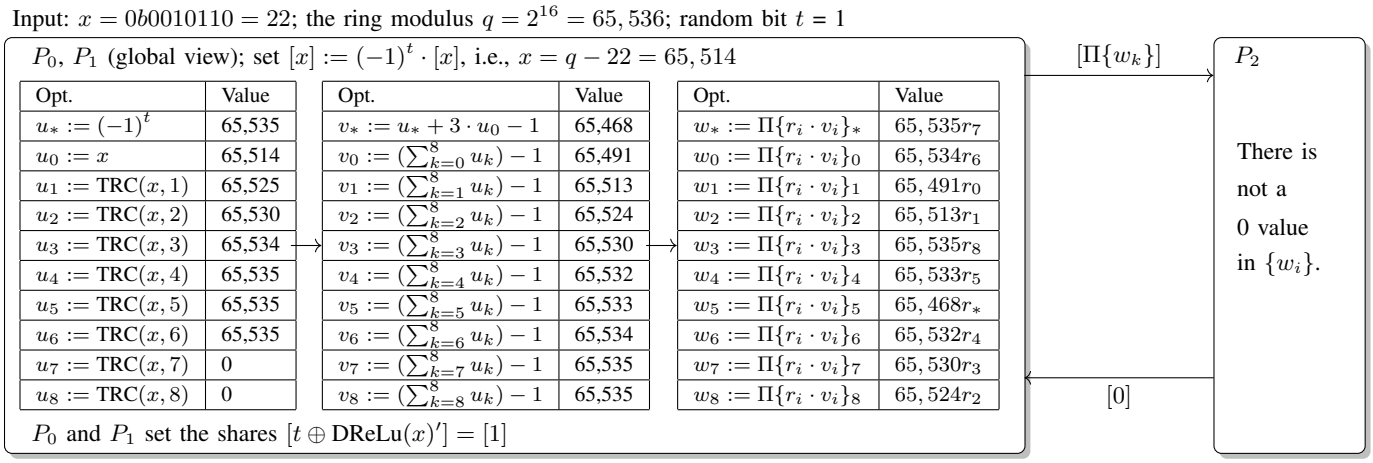


Figure 7: DReLU example for a positive input. (Opt.: operation, omitting the module operations. Val.: value.)

- If $\text{TRC}(\xi, \lambda' - 1) = 1$, then $\text{TRC}(\xi, \lambda') = 1$ or 0.
- If $\text{TRC}(q-\xi, \lambda'-1) = q-1$, then $\text{TRC}(q-\xi, \lambda') = q-1$ or 0.

Proof: **Case I**: For $\text{TRC}(\xi, \lambda' - 1) = 1$, we have $\text{TRC}(\xi, \lambda' - 1) = \frac{\xi}{2^{\lambda'-1}} + \mathsf{bit}_{\lambda'-1}$ according to Lemma 2, in which $\mathsf{bit}_{\lambda'-1} = 1$ or 0, with a probability of $1 - 2^{\lambda'-\ell}$. $\text{TRC}(\xi, \lambda' - 1) = 1$ implies two sub-cases.

- For $\frac{\xi}{2^{\lambda'-1}} = 1$ and $\mathsf{bit}_{\lambda'-1} = 0$, $\frac{\xi}{2^{\lambda'}} = 0$. Since $\frac{\xi}{2^{\lambda'-1}} = 1$, we have $\text{TRC}(\xi, \lambda') = \frac{\xi}{2^{\lambda'}} + \mathsf{bit}_{\lambda'} = \mathsf{bit}_{\lambda'}$, where $\mathsf{bit}_{\lambda'} = 0$ or 1. Hence, $\text{TRC}(\xi, \lambda') = 0$ or 1.
- For $\frac{\xi}{2^{\lambda'-1}} = 0$ and $\mathsf{bit}_{\lambda'-1} = 1$, $\frac{\xi}{2^{\lambda'}} = 0$. Since $\frac{\xi}{2^{\lambda'-1}} = 0$, we have $\text{TRC}(\xi, \lambda') = \frac{\xi}{2^{\lambda'}} + \mathsf{bit}_{\lambda'} = \mathsf{bit}_{\lambda'}$, where $\mathsf{bit}_{\lambda'} = 0$ or 1. Hence, $\text{TRC}(\xi, \lambda') = 0$ or 1.

Case II: For $\text{TRC}(q - \xi, \lambda' - 1) = q - 1$, we have $\text{TRC}(q-\xi, \lambda'-1) = q - \frac{\xi}{2^{\lambda'-1}} - \mathsf{bit}_{\lambda'-1}$, with a probability of $1-2^{\lambda'-\ell}$, according to Lemma 2. $\text{TRC}(q-\xi, \lambda'-1) = q-1$ implies two sub-cases, i.e., $\frac{\xi}{2^{\lambda'-1}} = 1$ and $\mathsf{bit}_{\lambda'-1} = 0$, and $\frac{\xi}{2^{\lambda'-1}} = 0$ and $\mathsf{bit}_{\lambda'-1} = 1$. Similar to the proof of the two sub-cases of Case I, these two sub-cases both satisfy that $\text{TRC}(q - \xi, \lambda') = q - 1$ or 0. □

Lemma 6. If $\ell > \lambda + 1$, for any number $\lambda' > \ell_x$, $\text{TRC}(x, \lambda') = 0$ if $\text{TRC}(x, \lambda'-1) = 0$, with a probability of $1 - 2^{\lambda'-\ell}$.

Proof: **Case I**: For $x = \xi$, $\text{TRC}(x, \lambda' - 1) = \frac{\xi}{2^{\lambda'-1}} + \mathsf{bit}_{\lambda'-1}$ according to Lemma 2, in which $\mathsf{bit}_{\lambda'-1} = 0$ or 1, with a probability of $1 - 2^{\lambda'-\ell}$. Under this condition, since $\text{TRC}(x, \lambda' - 1) = 0$, we have $\frac{\xi}{2^{\lambda'-1}} = 0$ and $\mathsf{bit}_{\lambda'-1} = 0$. Also, Lemma 2 implies $\lambda \leq \lambda' - 1$ and $r'_{\lambda'-1} \geq \xi'_{\lambda'-1}$. Hence, $r'_{\lambda'} = r_{\lambda'-1} \cdot 2^{\lambda'-1} + r'_{\lambda'-1}$ and $\xi'_{\lambda'} = \xi_{\lambda'-1} \cdot 2^{\lambda'-1} + \xi'_{\lambda'-1} = \xi_{\lambda'-1}$, where $r_{\lambda'-1}$ and $\xi_{\lambda'-1}$ are $(\lambda'-1)$-th bits of r and ξ respectively. Therefore, $r'_{\lambda'} \geq \xi'_{\lambda'}$. Then, $\text{TRC}(\xi, \lambda') = \frac{\xi}{2^{\lambda'}} + \mathsf{bit}_{\lambda'} = 0$, where $\frac{\xi}{2^{\lambda'}} = 0$ and $\mathsf{bit}_{\lambda'} = 0$, due to Lemma 2.

Case II: For $x = q - \xi$, $\text{TRC}(x, \lambda' - 1) = q - \frac{\xi}{2^{\lambda'-1}} - \mathsf{bit}_{\lambda'-1}$ according to Lemma 2, in which $\mathsf{bit}_{\lambda'-1} = 0$ or 1, with a probability of $1 - 2^{\lambda'-\ell}$. Under this condition, since $\text{TRC}(x, \lambda' - 1) = 0$, we have $\frac{\xi}{2^{\lambda'-1}} = 0$ and $\mathsf{bit}_{\lambda'-1} = 0$. Also, Lemma 2 implies $\lambda \leq \lambda' - 1$ and $r'_{\lambda'-1} + \xi'_{\lambda'-1} < 2^{\lambda'-1}$. Hence, $r'_{\lambda'} = r_{\lambda'-1} \cdot 2^{\lambda'-1} + r'_{\lambda'-1}$ and $\xi'_{\lambda'} = \xi_{\lambda'-1} \cdot 2^{\lambda'-1} + \xi'_{\lambda'-1} = \xi'_{\lambda'-1}$, where $r_{\lambda'-1}$ and $\xi_{\lambda'-1}$ are $(\lambda'-1)$-th bits of r and ξ respectively. Therefore, $r'_{\lambda'} + \xi'_{\lambda'} < 2^{\lambda'}$. Then, $\text{TRC}(x, \lambda') = \text{TRC}(q - \xi, \lambda') = q - \frac{\xi}{2^{\lambda'}} - \mathsf{bit}_{\lambda'} = 0$, where $\frac{\xi}{2^{\lambda'}} = 0$ and $\mathsf{bit}_{\lambda'} = 0$, due to Lemma 2. □

Finally, all the proofs of Lemma 1, Lemma 2, Lemma 3, Lemma 4, Lemma 5, and Lemma 6 imply Theorem 1.

Appendix D. DReLU Example

Fig. 7 shows a DReLU example for a positive input $x = 0b00010110 = 22$. In this case, we set the ring $\mathbb{Z}_q = [0, 65535], q = 65536, \ell = 16$. If P_0 and P_1 generate a random bit $t = 1$, they would reverse the sum of their input shares. Then, u_* would be $q - 1 = 65535, u_0 = q - x = 65,514$. After the repeated times of probabilistic truncation, the $\{u_i\}$ array is obtained. The errors occur at $\text{TRC}(x, 2)$, $\text{TRC}(x, 5)$, and $\text{TRC}(x, 6)$. In order to prevent the repeating $(q-1)$'s in $\{u_i\}$ leaking information about the input's range, P_0 and P_1 compute the summation of the subarrays of $\{u_i\}$ (excluding v_*) and then subtracting one from each summation. Next, P_0 and P_1 apply random maskings and a random shuffle using seed_{01}, leading to the shares of $\{w_i\}$. After reconstructing $\{w_i\}$, if P_2 finds out there is no 0's in the array, then the blinded input is negative, and P_2 sends the shares of $\text{DReLu}(x)' := 0$ to P_0 and P_1. Finally, P_0 and P_1 execute an XOR operation to obtain the shares of the output $\text{DReLu}(x) := \text{DReLu}(x)' \oplus t = 0 \oplus 1 = 1$.