USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

SEAF: Secure Evaluation on Activation Functions with Dynamic Precision for Secure Two-Party Inference

Hao Guo and Zhaoqian Liu, *The Chinese University of Hong Kong, Shenzhen;*
Ximing Fu, *Harbin Institute of Technology, Shenzhen; Pengcheng Laboratory; Key Laboratory of Cyberspace and Data Security, Ministry of Emergency Management;*
Zhusen Liu, *Hangzhou Innovation Institute of Beihang University*

https://www.usenix.org/conference/usenixsecurity25/presentation/guo-hao-seaf

This paper is included in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

SEAF: Secure Evaluation on Activation Functions with Dynamic Precision for Secure Two-Party Inference

Hao Guo[1†], Zhaoqian Liu[1†], Ximing Fu[2,3,4*], Zhusen Liu[5]
[1]The Chinese University of Hong Kong, Shenzhen
[2]Harbin Institute of Technology, Shenzhen
[3]Pengcheng Laboratory
[4]Key Laboratory of Cyberspace and Data Security, Ministry of Emergency Management
[5]Hangzhou Innovation Institute of Beihang University
guohao.g@outlook.com, zhaoqianliu@link.cuhk.edu.cn, fuximing@hit.edu.cn, zhusen_liu@163.com

Abstract

Secure evaluation of non-linear functions is one of the most expensive operations in secure two-party computation, particularly for activation functions in privacy preserving machine learning (PPML). This work introduces SEAF, a novel framework for efficient Secure Evaluation on Activation Functions. SEAF is based on the linear approximation approach, but enhances it by introducing two key innovations: Trun-Eq based interval test protocols and linear approximation with dynamic precision, which have the potential for broader applicability. Furthermore, we classify common activation functions into several categories, and present specialized methods to evaluate them using our enhanced techniques. Our implementation of SEAF demonstrates $3.5\times$ to $5.9\times$ speedup on activation functions Tanh and Sigmoid compared to SirNN (S&P'21). When applied on GELU, SEAF outperforms Iron (NeurIPS'22) by more than $10\times$ and Bolt (S&P'24) by up to $3.4\times$. For end-to-end secure inference on BERT, the original GELU accounts for 31.3% and 22.5% of the total runtime in Iron and Bolt, respectively. In contrast, our optimized GELU reduces these proportions to 4.3% and 9.8%, eliminating GELU as a bottleneck in secure inference.

1 Introduction

With growing concerns over privacy and security, privacy preserving machine learning (PPML) has gained significant attention in recent years. PPML facilitates distributed machine learning while ensuring privacy, enabling multiple participants to collaboratively train a model without exposing their individual datasets. In secure inference, a typical scenario involves a client holding a private input x and a server possessing a pre-trained model $\mathcal{M}$. The client seeks to obtain the inference result $\mathcal{M}(x)$ without revealing x, such that the server maintains the privacy of the model. To implement PPML, the process of plaintext machine learning usually be followed, replacing the underlying operations with their secure counterparts, such as secure multiplication, secure comparison, and secure evaluation of non-linear functions. The primary challenge in implementing PPML lies in designing efficient protocols for these sub-operations.

Secure multi-party computation (MPC) [8, 24, 25] is a fundamental cryptographic technology proposed by Yao, enabling multiple participants to jointly evaluate a function without revealing their inputs. Therefore MPC is particularly well-suited for designing underlying protocols for PPML. Several early secure computation or PPML frameworks such as ABY [5], ABY2.0 [19], MiniONN [15] and SecureML [17] are entirely based on MPC. These frameworks use Beaver's triples [3] for multiplication or convolution operations and Garbled Circuits (GC) [24] technology to implement the non-linear operations. Recently, Homomorphic Encryption (HE) [22] has been employed for multiplication in linear layers, reducing communication costs and improving efficiency. However, HE remains inefficient for evaluating non-linear functions. Although GC can evaluate arbitrary functions, it suffers from high communication and computational costs. Thus, there is a pressing need for efficient MPC-based techniques for evaluating non-linear functions.

A general approach for evaluating non-linear functions is to decompose them into underlying sub-functions, which are then combined. For instance, evaluating $\mathsf{Sigmoid}(x) = \frac{1}{1-e^{-x}}$ involves computing e^{-x} and the reciprocal operation. However, these sub-functions are also non-linear and difficult to evaluate. To address this, approximation methods are employed. Disregarding the details of the non-linear function, we regard it as a curve, and use polynomial functions or piecewise linear functions to fit it. Although the overhead of this method is independent from the complexity of the function, it still suffers from some inefficiencies. For instance, when evaluating non-linear function such as Tanh, Sigmoid or GeLU, the initial step typically involves determining whether the input lies within the non-linear or linear interval. Most existing works, such as MiniONN [15], CipherGPT [11] and Bolt [18], rely on two comparison protocols to identify the interval of

[†]Hao Guo and Zhaoqian Liu contributed equally to this work.
[*]Ximing Fu is the corresponding author.

the input, which consequently introduces significant overhead. In this work, we focus on enhancing the piecewise linear approximation method by introducing several novel techniques, which we believe have the potential for broader application in various contexts. Our primary contributions are as follows:

- We propose a series of novel interval test protocols based on the Trun-Eq approach, outperforming prior approach by 1.2× to 1.97×.
- We introduce dynamic precision method to evaluate non-linear functions, achieving both high accuracy and reduced overhead.
- We introduce SEAF, a general framework for efficiently evaluating common activation functions. Our Implementation demonstrates significant improvements across several widely-used activation functions and the end-to-end secure inference task.

The details of our main contributions are listed below.

Trun-Eq based interval test protocols. Our interval test protocol is inspired by the observation that determining whether $x \in [0, 2^h)$ is equivalent to checking if $\lfloor \frac{x}{2^h} \rfloor = 0$. This insight enables implementation of a single interval test protocol by combining a truncate-and-reduce protocol with a zero-checking protocol. Compared to prior approach relying on two comparison protocols in Bolt [18], this single interval test protocol reduces communication costs by 41.6%, and improves runtime performance by 1.66× to 1.97×. Based on this protocol, we also propose extended interval test protocols for more complex application scenarios, which outperform prior approach by 1.2× to 1.82×.

Evaluating non-linear functions with dynamic precision. In the linear approximation method, the overhead depends on the bitwidths of the slopes and intercepts of the linear functions, while the accuracy is determined by their values. We observe that these parameters typically fall within a small range, enabling significant bitwidth reductions. We further employ dynamic precision, optimizing the bitwidth to minimize overhead while ensuring that the approximation error remains bounded by a predefined threshold ε. Compared to prior approaches that uses fixed precision, the dynamic precision method can reduce the communication of the linear approximation by approximately 29% to 48%, while maintaining high accuracy.

Implementation. Based on these optimizations, we propose SEAF, a general framework for efficiently evaluating activation functions. The overview of SEAF is shown in Figure 1. We implemented SEAF on several popular activation functions, including ELU, Tanh, Sigmoid and GeLU. Compared to SirNN [20], SEAF achieves 3.5× to 5.9× improvement on Tanh and Sigmoid. For GELU, SEAF outperforms Iron's work [10] by more than 10×, and outperforms Bolt's work [18] by up to 3.4×, while providing comparable or superior accuracy. Specifically, the maximum ULP (Unit in the Last Place) error of Bolt's GELU is 14, and our high accuracy GELU can reduce this error to 3, while it still has a 1.8× to 2.7× improvement in runtime.

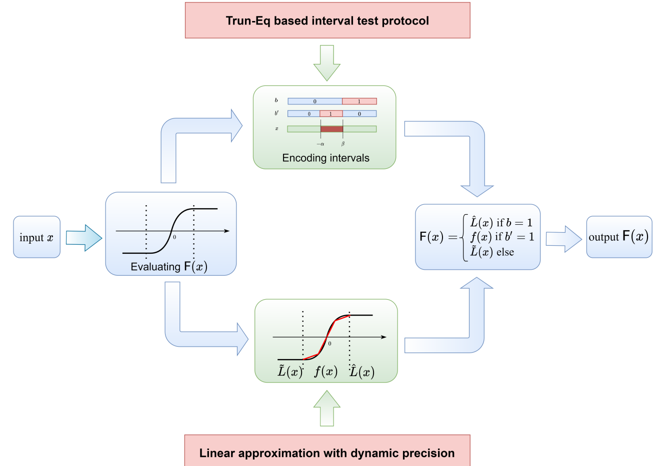


Figure 1: Overview of SEAF.

In the end-to-end benchmark, our optimized GELU reduces communication in secure inference on BERT [6] by 20%, with a 1.16× runtime improvement compared to Bolt. Compared to Iron [10], our approach enhances the efficiency of secure inference task by 1.39×. Additionally, in the original Iron and Bolt, the GELU accounts for 31.3% and 22.5% of the total runtime, respectively. By employing our optimized GELU, these proportions are significantly reduced to 4.3% and 9.8%, effectively eliminating GELU as a bottleneck in secure inference on BERT.

1.1 Techniques Overview

This work uses the linear approximation method [11, 15] to evaluate non-linear activation functions, while introducing new sub-protocols and methods to enhance its efficiency. Most activation functions exhibit strong linearity for relatively large or small input values, with non-linearity in a specific interval $[\alpha, \beta)$. Consequently, a non-linear activation function AF can be approximated as:

$$\mathsf{AF}(x) \approx \mathsf{F}(x) = \begin{cases} \hat{L}(x), & \text{if } x < \alpha \\ f(x), & \text{if } \alpha \leq x < \beta \\ \tilde{L}(x), & \text{if } x \geq \beta \end{cases} \quad (1)$$

where $f(x)$ is non-linear function, while $\hat{L}$ and $\tilde{L}$ are linear or constant functions. The non-linear interval $[\alpha, \beta)$ is then divided into smaller sub-intervals, within which $f(x)$ is ap-

proximated by linear functions:

$$f(x) \approx \begin{cases} l_0(x), & \text{if } \gamma_0 \le x < \gamma_1 \\ l_1(x), & \text{if } \gamma_1 \le x < \gamma_2 \\ \dots \\ l_{n-1}(x), & \text{if } \gamma_{n-1} \le x < \gamma_n \end{cases} \quad (2)$$

where $\gamma_0 = \alpha$, $\gamma_n = \beta$ and $l_i(x) = a_i x + d_i$ for $i = 0, ..., n-1$ are linear functions. Using $f(x)$, $\hat{L}(x)$, and $\tilde{L}(x)$, the overall approximation $\mathsf{F}(x)$ is computed as an approximation of $\mathsf{AF}(x)$.

We propose the SEAF framework to efficiently evaluate common activation functions using improved linear approximation method. The overview of SEAF is shown in Figure 1. Similar to prior works in MiniONN [15] and CipherGPT [11], SEAF evaluates activation functions in three steps: (1) determining the interval x falls into, (2) computing $f(x)$, $\hat{L}(x)$ and $\tilde{L}(x)$, and (3) compute $\mathsf{F}(x)$ from $f(x)$, $\hat{L}(x)$ and $\tilde{L}(x)$. To enhance efficiency, SEAF employs a novel Trun-Eq based interval test protocol and dynamic precision method. These innovations significantly reduce the cost of evaluating activation functions. These technologies are described below.

1.1.1 Trun-Eq based interval test protocol

We begin with a simple single interval test problem that determines whether $x \in [0, 2^h)$. A straightforward approach involves invoking two DReLU protocols to determine whether $x \ge 0$ and $x < 2^h$, requiring $2\lambda(l-1) + 28(l-1)$ bits of communication for an l-bit input. This method is utilized in many frameworks such as MiniONN [15], CipherGPT [11] and Bolt [18]. However, observing that $x \in [0, 2^h)$ is equivalent to $\lfloor \frac{x}{2^h} \rfloor = 0$, we can use a truncate-and-reduce protocol (with l-bit input) followed by a checking zero protocol (with $l-h$ bits input) instead. Intuitively, the truncate-and-reduce operation maps the interval $[0, 2^h)$ to a point 0, and the checking zero operation verifies whether x has been mapped to zero, as shown in Figure 2. The total communication of this method is $\frac{3}{4}\lambda l + \frac{1}{4}\lambda h$, which is comparable to a single DReLU protocol. DReLU is the derivative of ReLU and defined as that $\mathsf{DReLU}(x) = 1$ if $x \ge 0$ and $\mathsf{DReLU}(x) = 0$ otherwise.

To evaluate $\mathsf{F}(x)$ in Equation 1, additional checks for $x < \alpha$ and $x \ge \beta$ are required. We propose a new DReLU and checking -1 protocol to determine whether $x \ge 0$ and $x = -1$ simultaneously. Based on this sub-protocol, we design several new interval test protocols for different α and β, with similar or slightly larger overhead than a single DReLU protocol. The details of our new interval test protocols are presented in Section 3.

1.1.2 Evaluating $f(x)$ with dynamic precision

The overhead of evaluating non-linear $f(x)$ is primarily determined by the bitwidths of the slope a_i and intercept d_i. Prior approach typically sets these bitwidths to the same as the

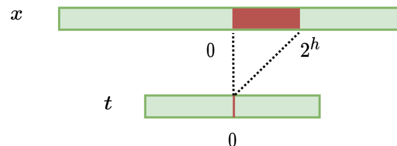


Figure 2: Idea of single interval test.

input x's bitwidth l. However, we observe that the values of a_i and d_i lie within a small range $(-1, 1)$ for most common activation functions. Therefore, only their fractional parts need to be represented, and the bitwidths are determined by their precisions , denoted as f_a and f_d. The accuracy of evaluating $f(x)$ depends on the values of the slopes and intercepts, which are influenced by f_a and f_d. Thus, the challenge becomes optimizing f_a and f_d to minimize overhead, and setting the values of slopes and intercepts to achieve high accuracy. We formulate this as an optimization problem, minimizing an overhead function subject to an error constraint. By solving this model, we can get the optimal precisions, values of slopes and intercepts for approximating $f(x)$. This dynamic precision method allows us to achieve both low overhead and high accuracy. Additionally, we expect this method also to be beneficial for evaluating general non-linear functions.

1.2 Related Work

Activation functions are crucial components in machine learning, and their secure evaluation often constitutes significant overhead in PPML frameworks. Early PPML frameworks, such as SecureML [17], employed garbled circuits technology to evaluate complex activation functions, introducing significant overhead. CrypTFlow2 [21] proposed an efficient comparison protocol, enabling the development of a new protocol for secure ReLU computation. For evaluating complex activation functions such as Tanh, Sigmoid, Sirnn [20] follow plaintext procedures, designing new sub-protocols such as exponential and reciprocal operations to evaluate them. While these methods achieve high accuracy, they incur substantial overhead. For highly complex functions like GeLU, the large communication overhead renders these methods nearly impractical.

Non-linear activation functions can be viewed as curves and approximated using (piecewise) polynomial functions. Bolt [18] utilized fourth and fifth degree polynomials to approximate GeLU and Tanh, respectively. Similarly, BumbleBee [16] and PUMA [7] employed piecewise polynomial fitting for GeLU. In contrast, MiniONN [15] and CipherGPT [11] adopted piecewise linear approximation, avoiding multiple calls to multiplication protocols. Since it requires querying lookup tables to retrieve the secret slope and intercept, it still faces inefficiency.

2 Preliminary

2.1 Notation

This work uses the notation $[\![x]\!]^l$ to denote a shared value x over the ring $\mathbb{Z}_{2^l}$, and $[\![\cdot]\!]^B$ denote a boolean sharing over $\mathbb{Z}_2$. $\mathbf{1}\{b\}$ denotes the indicator function that is 1 when b is true, and 0 otherwise. An element $x \in \mathbb{Z}_{2^l}$ represents an unsigned value, denoted as $\mathsf{uint}(x)$. The signed value of x, denoted $\mathsf{int}(x)$, is computed as $\mathsf{int}(x) = \mathsf{uint}(x) - \mathsf{MSB}(x) \cdot 2^l$, where $\mathsf{MSB}(x) = \mathbf{1}\{x \geq 2^{l-1}\}$. The notation $x \gg k$ represents truncating x by k bits, and $\lfloor \cdot \rfloor$ denotes the floor function. For a bit b, $\neg b$ means $b \oplus 1$. The symbol $[n]$ refers to the set $\{0, 1, \ldots n-1\}$. λ is the security parameter and typically set to 128.

2.2 Cryptographic Primitives

Fixed-point representation. Cryptographic operations are performed over rings or fields. In this work, we encode real numbers on the ring $\mathbb{Z}_{2^l}$ using a fixed-point representation. As OT-based protocols on ring $\mathbb{Z}_{2^l}$ can perform better than on the prime [12]. In this context, a real number $\hat{x} \in \mathbb{R}$ is encoded as an l-bit $x \in \mathbb{Z}_{2^l}$, where the first $l - f$ bits represent the integer part, and the remaining f bits represent the fractional part, also known as the precision. $\hat{x} \in \mathbb{R}$ and its fixed-point representation $x \in \mathbb{Z}_{2^l}$ can be calculated by each other as $x = \lfloor \hat{x} \cdot 2^f \rfloor \mod 2^l$ and $\hat{x} = \frac{\mathsf{int}(x)}{2^f}$.

Secret sharing. This work employs the 2-out-of-2 additive secret sharing schemes over different power-of-2 rings. A shared $x \in \mathbb{Z}_{2^l}$ is denoted as $[\![x]\!]^l = ([\![x]\!]^l_0, [\![x]\!]^l_1)$, such that $x = [\![x]\!]^l_0 + [\![x]\!]^l_1 \mod 2^l$, where P_i holds $[\![x]\!]^l_i$ and can not get any information about $[\![x]\!]^l_{1-i}$ for $i \in \{0,1\}$. For simplicity, in this work we abbreviate $[\![x]\!]^l_i$ as x_i for $i \in \{0,1\}$. When $l \geq 2$, the shares $[\![x]\!]^l$ are referred to as arithmetic shares; when $l = 1$, they are called boolean shares and denoted as $[\![x]\!]^B$.

Oblivious transfer. Oblivious Transfer (OT) is a very fundamental component in MPC. Let $\binom{k}{1}$-OT_l denote a 1-out-of-k OT protocol. In this protocol, the sender inputs k l-bit messages $m_0, \ldots, m_{k-1}$ and the receiver inputs a choice value $j \in [k]$. Then the receiver gets output m_j for $j \in [k]$ while he has no idea of other messages, and the sender receives no output. The $\binom{k}{1}$-OT_l is generalized from the 1-out-of-2 OT using the OT extension technology [13, 14]. Additionally, we also use the correlated 1-out-of-2 OT, denoted as $\binom{2}{1}$-COT_l, where the sender inputs an element $x \in \mathbb{Z}_{2^l}$ and the receiver inputs a choice bit b. The sender then outputs a random $r \in \mathbb{Z}_{2^l}$, while receiver outputs either r or $x + r$, depending on the value of b. These OTs can be implemented using either IKNP-style [13] or VOLE-style OT [4, 23]. For IKNP-style OT, the communication for $\binom{k}{1}$-OT_l and $\binom{2}{1}$-COT_l is $2\lambda + kl$ and $\lambda + l$, respectively. Specifically, when $k = 2$, the communication for $\binom{2}{1}$-OT_l is $\lambda + 2l$ bits. In contrast, VOLE-style OT achieves significantly lower communication overhead but introduces higher computational costs. Throughout this work, unless stated otherwise, we adopt IKNP-style OT for protocol implementation.

Table 1: Descriptions of functionalities used in this work.

Functionalities	Outputs
$\mathcal{F}_{\mathsf{AND}}([\![x]\!]^B, [\![y]\!]^B)$	outputs $[\![z]\!]^B$ satisfies $z = x \wedge y$
$\mathcal{F}_{\mathsf{MUX}}([\![x]\!]^l, [\![b]\!]^B)$	outputs $[\![z]\!]^l$ satisfies $z = x$ if $b = 1$, else $z = 0$.
$\mathcal{F}^l_{\mathsf{SS}}([\![x]\!]^l, [\![y]\!]^l, [\![b]\!]^B)$	outputs $[\![z]\!]^l$ satisfies $z = x$ if $b = 1$, else $z = y$.
$\mathcal{F}_{\mathsf{LUT}}(T, [\![I]\!]^m)$	outputs $[\![T[I]]\!]^n$.
$\mathcal{F}^l_{\mathsf{Mill}}(x, y)$	outputs $[\![b]\!]^B$ satisfies $b = \mathbf{1}\{x < y\}$.
$\mathcal{F}^l_{\mathsf{DReLU}}([\![x]\!]^l)$	outputs $[\![b]\!]^B$ satisfies $b = \mathbf{1}\{x \geq 0\}$.
$\mathcal{F}^l_{\mathsf{Eq}}(x, y)$	outputs $[\![b]\!]^B$ satisfies $b = \mathbf{1}\{x = y\}$.
$\mathcal{F}^l_{\mathsf{Zero}}([\![x]\!]^l)$	outputs $[\![b]\!]^B$ satisfies $b = \mathbf{1}\{x = 0\}$.
$\mathcal{F}^l_{\mathsf{Comp\&Eq}}(x, y)$	outputs $[\![b]\!]^l$ and $[\![b^*]\!]^B$ satisfy $b = \mathbf{1}\{x < y\}$ and $b^* = \mathbf{1}\{x = y\}$.
$\mathcal{F}^k_{\mathsf{Trun}}([\![x]\!]^l)$	outputs $[\![z]\!]^l$ satisfies $z = x \gg k$.
$\mathcal{F}^k_{\mathsf{AppTrun}}([\![x]\!]^l)$	outputs $[\![z]\!]^l$ satisfies $z = [\![x \gg k + \delta]\!]^l$, $\delta \in \{0,1\}$.
$\mathcal{F}^k_{\mathsf{TR}}([\![x]\!]^l)$	outputs $[\![z]\!]^{l-k}$ satisfies $z = x \gg k$.
$\mathcal{F}^{m,n}_{\mathsf{SExt}}([\![x]\!]^m)$	outputs $[\![x]\!]^n$ where $m < n$.
$\mathcal{F}^{m,n}_{\mathsf{SMul}}([\![x]\!]^m, [\![y]\!]^n)$	outputs $[\![z]\!]^{m+n}$ satisfies $z = xy$.

2.3 2PC Functionalities

We denote two-party functionality for computing a function func as $\mathcal{F}_{\mathsf{func}}$. The statement "$P_0$ and P_1 invoke $\mathcal{F}_{\mathsf{func}}(x, y)$ (resp. $\mathcal{F}_{\mathsf{func}}([\![x]\!])$) to learn $[\![z]\!]$" means that P_0 with input x (resp. $[\![x]\!]_0$) and P_1 with input y (resp. $[\![x]\!]_1$) invoke $\mathcal{F}_{\mathsf{func}}$ and learn $[\![z]\!]$. The functionality is in the ideal world, and to implement it in real world, the corresponding protocol Π_{func} is invoked. The 2-party functionalities used in this work are summarized in Table 1, and the corresponding protocols and their overheads are listed below.

AND. The Π_{AND} computes the logical AND of two bits a and b. It takes Boolean shares $[\![a]\!]^B$ and $[\![b]\!]^B$ as inputs, and returns $[\![c]\!]^B$ such that $c = a \wedge b$. Π_{AND} can be implemented using Beaver bit triples [21], with an amortized cost of $\lambda + 20$ bits per AND gate.

Multiplexer (MUX). The multiplexer protocol Π^l_{MUX} takes arithmetic shares $[\![x]\!]^B$ and Boolean shares $[\![b]\!]^B$ as input, and outputs arithmetic shares $[\![y]\!]^B$ where $y = \mathsf{MUX}(x, b) = x \cdot b$.

Π^l_{MUX} can be implemented using two calls to COT [20], with communication $2(\lambda + l)$ in 2 rounds.

Select shares (SS). We define a select shares protocol Π^l_{SS}, which takes $[\![x]\!]^l$, $[\![y]\!]^l$ and $[\![b]\!]^B$ as inputs, and outputs $[\![x]\!]^l$ if $b = 1$, and $[\![y]\!]^l$ otherwise. This protocol can be implemented by invoking one Π^l_{MUX} as $\mathsf{SS}(x,y,b) = \mathsf{MUX}(x-y,b)+y$. Further, if $y = -x$ then Π^l_{SS} outputs $|x|$.

Lookup table (LUT). The lookup table protocol Π_{LUT} takes a public lookup table T and an shared index $[\![I]\!]^l$ as input, and outputs $[\![T[I]]\!]^l$. For table T with M entries of n-bit each, this protocol can be realized using a single call to $\binom{M}{1}$-OT$_n$, with communication $2\lambda + Mn$ in 2 rounds.

Millionaires's protocol (Mill). The Millionaires's protocol Π^l_{Mill} also known as comparison protocol, computes a Boolean value $[\![b]\!]^B$ such that $b = \mathsf{Mill}(x,y) = \mathbf{1}\{x < y\}$, with x and y being l-bit numbers held by P_0 and P_1, respectively. CrypTFlow2 [21] provides an efficient implementation of Π^l_{Mill}, with communication less than $\lambda l + 14l$ bits in $\log l$ rounds.

DReLU. Π^l_{DReLU} takes $[\![x]\!]^l$ as input, and outputs $[\![b]\!]^B$ where $b = \mathsf{DReLU}(x) = 1$ if $x \geq 0$, and $b = 0$ otherwise. Π^l_{DReLU} can be realized by invoking one $\Pi^{l-1}_{\mathsf{Mill}}$ [21], with communication less than $\lambda(l-1) + 14(l-1)$.

Equality (Eq). The equality test protocol Π^l_{Eq} checks whether two values are equal. P_0 inputs an l-bit x and P_1 inputs an l-bit y, then Π^l_{Eq} returns $[\![b]\!]^B$, such that $b = \mathsf{Eq}(x,y) = \mathbf{1}\{x = y\}$. Π^l_{Eq} can be implemented using the idea of the Millionaires's protocol in CrypTFlow2 [21], with communication less than $\frac{3}{4}\lambda l + 4l$ in $\log l$ rounds.

Check zero (Zero). The check zero protocol Π^l_{Zero} takes $[\![x]\!]^l$ as input and outputs $[\![b]\!]^B$ such that $b = \mathbf{1}\{x = 0\}$. Π^l_{Zero} can be realized by invoking a Π^l_{Eq}, where P_0 inputs x_0 and P_1 inputs $2^l - x_1 \mod 2^l$, with the same overhead as Π^l_{Eq}.

Compare and equal (Comp&Eq). Building on the work of CrypTFlow2 [21], the protocols Π^l_{Mill} and Π^l_{Eq} can be implemented simultaneously. SirNN [20] proposed the compare and equal protocol $\Pi^l_{\mathsf{Comp\&Eq}}$, where P_0 inputs an l-bit x and P_1 inputs an l-bit y, then they learn $[\![b]\!]^B$ and $[\![b^*]\!]^B$ such that $b = \mathbf{1}\{x < y\}$ and $b^* = \mathbf{1}\{x = y\}$. The communication of $\Pi^l_{\mathsf{Comp\&Eq}}$ is less than $\lambda l + 14l$ in $\log l$ rounds.

Truncation (Trun). Truncation protocol Π^k_{Trun} is used to truncate the last k-bit of the input, which takes $[\![x]\!]^l$ as input and output $[\![x \gg k]\!]^l$. Additionally, for the case one-bit error is tolerated, we can instead use the one-bit error truncation protocol Π^k_{AppTrun}, which outputs $[\![x \gg k]\!]^l + \delta$ where $\delta \in \{0,1\}$. Using protocols in SirNN [20] and Cheetah [12], the communication of Π^k_{Trun} and Π^k_{AppTrun} is $\lambda(l+3) + 15l + k + 20$ and $\lambda(l+1) + 14l + k$, respectively. However, if $\mathsf{MSB}(x)$ is known in public, their communication can be reduced to $\lambda(k+3) + 15k + l + 2$ and $2\lambda + k + 2$.

Truncation-and-reduce (TR). The truncation-and-reduce protocol Π^k_{TR} is also used to truncate k-bit of input x, while different from $\Pi^{l,k}_{\mathsf{Trun}}$, it output $[\![x \gg k]\!]^{l-k}$ but not $[\![x \gg k]\!]^l$. This work implements SirNN's $\Pi^{l,k}_{\mathsf{TR}}$, with communication less than $\lambda(k+1) + 15k$ in $\log k + 2$ rounds.

Signed extension (SExt). The signed extension protocol $\Pi^{l,l'}_{\mathsf{SExt}}$ takes $[\![x]\!]^l$ as input, and outputs $[\![x]\!]^{l'}$ for $l' > l$. We implement $\Pi^{l,l'}_{\mathsf{SExt}}$ using the method in SirNN [20], with communication $\lambda(l+1) + 13l + l'$. Further, if $\mathsf{MSB}(x)$ is known, this communication can be reduced to $2\lambda + l - l' + 2$.

Signed non-uniform multiplication (SMul). The signed non-uniform multiplication $\Pi^{m,n}_{\mathsf{SMul}}$ takes $[\![x]\!]^m$ and $[\![y]\!]^n$ as input, and outputs $[\![z]\!]^{m+n}$ satisfies $z = xy$. The work implement $\Pi^{m,n}_{\mathsf{SMul}}$ using the method in SirNN [20], with communication $3\lambda(\mu + \nu) + (m+n)^2 + 15(m+n)$, where $\mu = \min\{m,n\}$ and $\nu = \max\{m,n\}$. Further, if $\mathsf{MSB}(x)$ and $\mathsf{MSB}(y)$ are known, this communication can be reduced to $(2\mu + 6)\lambda + \mu^2 + \mu + 2(mn + m + n)$.

2.4 Threat Model

We consider the security of two-party computation within the simulation paradigm, specifically against a static semi-honest probabilistic polynomial-time (PPT) adversary $\mathcal{A}$. A semi-honest adversary $\mathcal{A}$ is computationally bounded and corrupts one of the parties at the beginning of the protocol. Although the corrupted party follows the protocol honestly, the adversary attempts to infer additional information about the input of the honest party. To define the security of a protocol for computing any function f, we define two types of interactions: the real interaction and the ideal interaction. In the real interaction, parties P_0 and P_1 execute the protocol in the presence of an adversary $\mathcal{A}$ and an environment $\mathcal{Z}$. In the ideal interaction, P_0 and P_1 submit their inputs to a trusted functionality $\mathcal{F}$, which computes f and returns the outputs to the respective parties faithfully. Security requires that for every adversary $\mathcal{A}$ in the real interaction, there exists a corresponding adversary $\mathcal{S}$ (referred to as the simulator) in the ideal interaction, such that no environment $\mathcal{Z}$ can distinguish between the real and ideal interactions. Many of our protocols involve multiple sub-protocols, which we describe using the hybrid model. In this model, the interaction is similar to the real interaction, except that sub-protocols are replaced by calls to instances of the corresponding functionalities. A protocol that invokes a functionality $\mathcal{F}$ is said to operate in the "$\mathcal{F}$-hybrid model."

2.5 Common Activation Functions

Activation functions are essential components in machine learning. In this subsection we introduce several commonly used activation functions. Some other common activation functions are listed in Appendix A.

ELU. The Exponential Linear Unit (ELU) is defined as $\mathsf{ELU}(x) = \max\{0,x\} + \min\{0, \alpha \cdot (e^x - 1)\}$, where α is a hyperparameter typically set to 1. Unlike ReLU, ELU outputs non-zero gradients when the input is negative, addressing the gradient vanishing problem during training. Additionally, compared to ReLU, ELU can converge faster.

Tanh and Sigmoid. Activation functions such as Tanh and Sigmoid are commonly used in recurrent neural networks (RNNs), such as Google-30. These functions are defined as $\mathsf{Tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ and $\mathsf{Sigmoid}(x) = \frac{1}{1+e^{-x}}$. It is obvious that Tanh is an odd function. Moreover, since $1 - \mathsf{Sigmoid}(x) = \mathsf{Sigmoid}(-x)$, we define a shifted version $\mathsf{Sigmoid}^*(x) = \mathsf{Sigmoid} - \frac{1}{2}$, which is also an odd function. We can easily recover $\mathsf{Sigmoid}(x)$ from $\mathsf{Sigmoid}^*(x)$.

GELU. The Gaussian Error Linear Unit (GELU) is widely used in large language models such as BERT [6], and is defined as $\mathsf{GELU}(x) = \frac{1}{2}x \cdot (1 + \mathsf{erf}(\frac{x}{\sqrt{2}}))$, where the Gauss error function is $\mathsf{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$. Following Bolt [18], we define $g(x) = \frac{1}{2}x \cdot \mathsf{erf}(\frac{x}{\sqrt{2}})$, so that $\mathsf{GELU}(x) = \frac{1}{2}x + g(x)$. The function $g(x)$ exhibits good linearity for relatively large or small inputs, with the limits $\lim_{x \to +\infty} g(x) = \frac{x}{2}$ and $\lim_{x \to -\infty} g(x) = -\frac{x}{2}$. Additionally, $g(x)$ is an even function, making it easier to evaluate compared to the original GELU as $g(-x) = g(x)$ and only the not-negative input needs to be evaluated.

3 Trun-Eq based Interval Test Protocol

The first step of evaluating $\mathsf{F}(x)$ in Equation 1 is to determine the interval x falls into. In this section, we first classify the common activation functions into several categories. Then for each category, we propose new efficient interval test protocols to identify the interval in which x lies. Additionally, we outline the specific methods for evaluating $\mathsf{F}(x)$ in each category.

3.1 Classification of Activation Functions

Most common activation functions take the form of Equation 1, and can be considered non-linear only within the non-linear interval $[\alpha, \beta)$. Based on the non-linear interval and the function's parity, we classify these functions into the following categories. Specifically, a function $f(x)$ is an odd function if $f(x) = -f(-x)$, and it is even if $f(x) = f(-x)$. The details of activation functions listed in this subsection can be seen in Appendix A.

Category 1. $\alpha = \beta$. Functions in this category have no non-linear interval, and can be viewed as the concatenation of two linear functions. To evaluate $\mathsf{F}(x)$, we need to compute $b = \mathbf{1}\{x \geq \alpha\}$, and then $\mathsf{F}(x) = (\tilde{L}(x) - \hat{L}(x)) \cdot b + \hat{L}(x) = \mathsf{MUX}(\tilde{L}(x) - \hat{L}(x), b) + \hat{L}(x)$. This category includes activation functions such as ReLU, LeakyReLU, PReLU and so on.

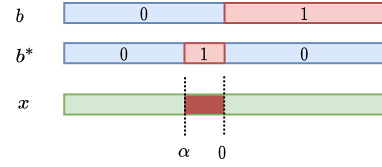


Figure 3: Interval segmentation for activation functions in Category 2.

Category 2. $\alpha < 0$ and $\beta = 0$. For functions in this category, we introduce two variables to encode the non-linear and two linear intervals as follows:

$$b = \mathbf{1}\{x \geq 0\}, \quad b^* = \mathbf{1}\{x \in [\alpha, 0)\}. \tag{3}$$

The values of b and b^* are shown in Figure 3, where the brown interval denotes the non-linear interval. Then $\mathsf{F}(x)$ can be computed as:

$$\begin{aligned} \mathsf{F}(x) &= f(x) \cdot b^* + \tilde{L}(x) \cdot b + \hat{L}(x) \cdot (\neg b \wedge \neg b^*) \\ &= \hat{L}(x) + ((f(x) - \hat{L}(x)) \cdot b^* + ((\tilde{L}(x) - \hat{L}(x)) \cdot b. \end{aligned} \tag{4}$$

This expression reduces the number of multiplications from three to two. Instances of activation functions in this category include ELU, CELU, and SELU. Moreover, if $f(x)$ is linear function, then Hardsigmoid, ReLU6 and Softshrink can also be classified in this category.

Category 3. $\alpha < 0$, $\beta = -\alpha$, and $f(x)$ is an odd or even function. In this category, the parities of functions are used. We divide the interval into four parts as Figure 4, and encode these intervals using three variables defined as:

$$b = \mathbf{1}\{x \geq 0\}, b^* = \mathbf{1}\{x \in [\alpha, 0)\}, b^\# = \mathbf{1}\{x \in [0, \beta)\}. \tag{5}$$

Then we can compute $\mathsf{F}(|x|)$ $\mathsf{F}(x)$ as follows:

$$\begin{cases} \mathsf{F}(|x|) = f(|x|) \cdot (b^\# \oplus b^*) + \tilde{L}(x) \cdot (b \oplus b^\#), \\ \mathsf{F}(x) = \mathsf{SS}(\mathsf{F}(|x|), (-1)^p \cdot \mathsf{F}(|x|), b) \end{cases} \tag{6}$$

where $p \in \{0,1\}$ is a public value, with $p = 1$ for odd functions and $p = 0$ for even functions. Although an extra variable $b^\#$ needs to be computed, which introduces extra overhead, the size of non-linear interval is reduced by half as only $f(|x|)$ needs to be computed. Common activation functions like Tanh, Sigmoid, Softsign, and Hardtanh fit into this category. Additionally, functions like GELU can be computed using functions from this category (see Section 2.5 for details).

Category 4. $\alpha < \beta$ and $\hat{L}(x) = \tilde{L}(x) = L(x)$. In this category, the linear functions $\hat{L}(x)$ and $\tilde{L}(x)$ are identical. We represent the non-linear interval by computing $b = \mathbf{1}\{x \in [\alpha, \beta)\}$, then $\mathsf{F}(x)$ can be computed by:

$$\begin{aligned} \mathsf{F}(x) &= f(x) \cdot b + L(x) \cdot \neg b \\ &= (f(x) - L(x)) \cdot b + L(x). \end{aligned} \tag{7}$$

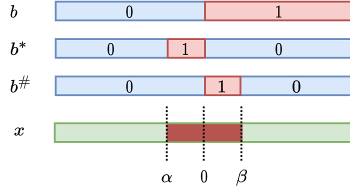


Figure 4: Interval segmentation for activation functions in Category 3.

Only some special activation functions such as Hardshrink and Tanhshrink can be classified in this category. However, in the training phase of ML, the derivatives of activation functions are used. Moreover, for most common activation functions, both the $\hat{L}(x)$ and $\tilde{L}(x)$ are constant functions. Then the derivative functions $\hat{L}'(x) = \tilde{L}'(x) = 0$, and $\mathsf{F}'(x)$ belongs to this category. Specifically, $\mathsf{F}'(x)$ can be evaluated as $\mathsf{F}'(x) = f'(x)$ for $x \in [\alpha, \beta)$, and $\mathsf{F}'(x) = 0$ otherwise, resulting in $\mathsf{F}'(x) = \mathsf{MUX}(f'(x), b)$.

This work focuses on secure inference task, so we primarily study how to evaluate the activation functions in Category 2 and Category 3. In the following subsections, we propose our new interval test protocols to compute the variables in Equation 3 and Equation 5.

3.2 Single Interval Test

In Section 1.1.1, we introduced the idea of the single interval test protocol $\Pi^l_{\mathsf{InSingle}}$. In this subsection, we provide the details of this protocol in Algorithm 1. $\Pi^l_{\mathsf{InSingle}}$ is used to determine whether an input belongs to a given interval $[\alpha, \beta)$, where we assume $\beta - \alpha = 2^h$. We first shift x to $y = x - \alpha$, converting the problem into determining whether $y \in [0, 2^h)$. We observe that $\mathbf{1}\{y \in [0, 2^h)\} = \mathbf{1}\{\lfloor \frac{y}{2^h} \rfloor = 0\}$, therefore a truncate-and-reduce protocol with h-bit inputs and a checking zero protocol with $l - h$ bits input are invoked, with total communication of $\frac{3}{4}\lambda l + \frac{1}{4}\lambda h + 4l + 10h$ bits. In contrast, traditional method [18] need to invoke two DReLU protocols with l-bit inputs, resulting in communication costs of $\lambda(l-1) + 14(l-1)$ bits. Using this single interval test protocol, we can compute the variable $b = \mathbf{1}\{x \in [\alpha, \beta)\}$ efficiently, and therefore efficiently evaluate functions in Category 4 following Equation 7.

3.3 Interval Test Protocol for Category 2

We now show how to compute the variables b and b^* defined in Equation 3. Similar to the single test protocol, we first suppose $\alpha = -2^h$ and truncate-and-reduce h-bit of x to get a $l - h$ bits t. Then the value of t can be written as:

$$\begin{cases} t < -1, & \text{if and only if} \quad x < \alpha \\ t = -1, & \text{if and only if} \quad x \in [\alpha, 0) \\ t \geq 0, & \text{if and only if} \quad x \geq 0 \end{cases}.$$

Thus, computing $b = \mathbf{1}\{x \geq 0\}$ and $b^* = \mathbf{1}\{x \in [\alpha, 0)\}$ is equivalent to computing $b_t = \mathbf{1}\{t \geq 0\}$ and $b^*_t = \mathbf{1}\{t = -1\}$. One straightforward approach would be to invoke a Π_{Zero} to check whether $t + 1 = 0$, and a Π_{DReLU} to compute $b_t = \mathbf{1}\{t \geq 0\}$. However, we find that these two protocols can be performed simultaneously, and we propose a combined DReLU and negone protocol $\Pi_{\mathsf{DReLU\&negone}}$ for efficiently computing both b and b^*.

Algorithm 1: Single interval test, $\Pi^l_{\mathsf{InSingle}}$:

Input: P_0 and P_1 hold $[\![x]\!]^l$, and public interval $[\alpha, \beta)$ where $\beta - \alpha = 2^h$.

Output: P_0 and P_1 output $[\![b]\!]^B$, such that $b = \mathbf{1}\{x \in [\alpha, \beta)\}$.

1. P_0 & P_1 compute $[\![y]\!]^l = [\![x]\!]^l - \alpha$.
2. P_0 & P_1 invoke $\mathcal{F}^h_{\mathsf{TR}}([\![y]\!]^l)$ to learn $[\![t]\!]^{l-h}$.
3. P_0 & P_1 invoke $\mathcal{F}^{l-h}_{\mathsf{Zero}}([\![t]\!]^{l-h})$ to learn $[\![b]\!]^B$.
4. P_0 & P_1 output $[\![b]\!]^B$.

3.3.1 DReLU and negone protocol

Suppose P_0 and P_1 hold $[\![x]\!]^l$, our goal is to design a protocol $\Pi^l_{\mathsf{DReLU\&negone}}$ which outputs $b = \mathsf{DReLU}(x)$ and $b^* = \mathbf{1}\{x = -1\}$ together. CrypTFlow2 [21] proposes an efficient method for computing $\mathsf{DReLU}(x)$. In their method, each party parses their input as $x_i = m_i || y_i$, where $i \in \{0,1\}$, $m_i \in \{0,1\}$, $y_i \in \{0,1\}^{l-1}$. Then $\mathsf{DReLU}(x)$ is computed as $\mathsf{DReLU}(x) = \mathsf{carry} \oplus m_0 \oplus m_1 \oplus 1$, where $\mathsf{carry} = \mathbf{1}\{y_0 + y_1 > 2^{l-1} - 1\}$, which is equivalent to computing $\mathsf{carry} = \mathbf{1}\{(2^{l-1} - 1 - y_0) < y_1\}$. Note that we can also invoke the $\Pi^{l-1}_{\mathsf{Comp\&Eq}}$ to compute carry, while it outputs $\mathsf{eq} = \mathbf{1}\{y_0 + y_1 = 2^{l-1} - 1\}$ additionally. Moreover, we give Theorem 1 to show that $\mathbf{1}\{\mathsf{int}(x) = -1\} = \mathbf{1}\{x_0 + x_1 = 2^l - 1\}$ can be computed from eq. Based on this theorem, we give the details of our $\Pi^l_{\mathsf{DReLU\&negone}}$ in Algorithm 2, with total communication less than $\lambda l + 14l$.

Theorem 1 *For $x = x_0 + x_1 \mod 2^l$, let $x_i = m_i || y_i$ where $m_i \in \{0,1\}$, $y_i \in \{0,1\}^{l-1}$ for $i \in \{0,1\}$. Define: $\mathsf{carry} = \mathbf{1}\{2^{l-1} - 1 - y_0 < y_1\}$, $\mathsf{eq} = \mathbf{1}\{2^{l-1} - 1 - y_0 = y_1\}$ and $\mathsf{eq}^* = \mathsf{eq} \wedge (m_0 \oplus m_1)$. Then $\mathsf{DReLU}(x) = m_0 \oplus m_1 \oplus \mathsf{carry} \oplus 1$ and $\mathsf{eq}^* = \mathbf{1}\{\mathsf{int}(x) = -1\}$.*

The proof of Theorem 1 is shown in Appendix B.

Algorithm 2: DReLU and equal to -1, $\Pi^l_{\mathsf{DReLU\&negone}}$:

Input: P_0 and P_1 hold $[\![x]\!]^l$.
Output: P_0 and P_1 output $[\![b]\!]^B$ and $[\![b^*]\!]^B$, where $b = \mathsf{DReLU}(x)$ and $b^* = \mathbf{1}\{\mathsf{int}(x) = -1\}$.

1. For $i \in \{0,1\}$, P_i parses $[\![x]\!]^l_i$ as $[\![x]\!]^l_i = m_i || y_i$, where $m_i \in \{0,1\}$ and $y_i \in \{0,1\}^{l-1}$.
2. P_0 & P_1 invoke $\mathcal{F}^{l-1}_{\mathsf{Comp\&Eq}}(2^{l-1} - 1 - y_0, y_1)$ to learn $[\![\mathsf{carry}]\!]^B$ and $[\![\mathsf{eq}]\!]^B$.
3. P_0 & P_1 compute $[\![b]\!]^B = [\![\mathsf{carry}]\!]^B \oplus m_0 \oplus m_1 \oplus 1$.
4. For $i \in \{0,1\}$, P_i sets $[\![m^*]\!]^B_i = m_i$.
5. P_0 & P_1 invoke Π_{AND} with inputs $[\![\mathsf{eq}]\!]^B$ and $[\![m^*]\!]^B$ to learn $[\![b^*]\!]^B$.
6. P_0 & P_1 output $[\![b]\!]^B$ and $[\![b^*]\!]^B$.

3.3.2 Interval test protocol

Based on $\Pi^l_{\mathsf{DReLU\&negone}}$, we propose the following interval test protocol Π^l_{InCate2} in Algorithm 3 to compute b and b^* defined in Equation 3. The total communication of Π^l_{InCate2} is bounded by $\lambda l + 14l$, which is slightly larger than the overhead of a single call to Π^l_{DReLU}.

Algorithm 3: Interval test for Category 2, Π^l_{InCate2}:

Input: P_0 and P_1 hold $[\![x]\!]^l$, and public $\alpha = -2^h$.
Output: P_0 and P_1 output $[\![b]\!]^B$ and $[\![b^*]\!]^B$ as defined in Equation 3.

1. P_0 & P_1 invoke $\mathcal{F}^h_{\mathsf{TR}}([\![x]\!]^l)$ to learn $[\![t]\!]^{l-h}$.
2. P_0 & P_1 invoke $\mathcal{F}^{l-h}_{\mathsf{DReLU\&negone}}([\![t]\!]^{l-h})$ to learn $[\![b]\!]^B$ and $[\![b^*]\!]^B$.
3. P_0 & P_1 output $[\![b]\!]^B$ and $[\![b^*]\!]^B$.

3.4 Interval Test Protocol for Category 3

Now we propose an interval test protocol Π^l_{InCate3} for computing b, b^* and $b^\#$ as defined in Equation 5, where we suppose $\beta = -\alpha = 2^h$. The values of b and b^* can be computed by invoking Π^l_{InCate2}. The value of $b^\#$ can be computed as $b^\# = \mathbf{1}\{x \in [0,\beta)\} = \mathbf{1}\{x \gg h = 0\}$, therefore an $\Pi^{l-h}_{\mathsf{Zero}}$ is required. The details of Π^l_{InCate3} are shown in Algorithm 4, with total communication less than $\frac{7l-3h}{4} \cdot \lambda + 14l - 4h$.

Discussion. The approach of Π_{InCate2} can also be adapted to implement Π_{InCate3}, but this introduces an additional error and requires extra overhead to eliminate it. Specifically, determining whether $x \in [\alpha,\beta)$ is equivalent to computing $\mathbf{1}\{x - \beta \in [-2\alpha, 0)\}$, which can be achieved by invoking Π_{InCate2}. However, all calculations are performed on the ring $\mathbb{Z}_{2^l}$, and when we compute $x - \beta$, we actually compute $(x-\beta)$ mod 2^l. Consequently, for negative x within the range $\mathsf{int}(x) \in [-2^{l-1}, -2^{l-1} + \beta)$, it will be shifted to the positive range as $(x-\beta) \mod 2^l \in [2^{l-1} - \beta, 2^{l-1})$. This shifting operation changes the sign of x, leading to $\mathsf{DReLU}(x) \neq \mathsf{DReLU}((x-\beta)$ mod $2^l)$. To resolve this issue, an additional $\Pi^{l-h}_{\mathsf{Zero}}$ is needed to check whether $x \in [-2^{l-1}, -2^{l-1} + \beta)$. As a result, the overhead of the modified Π_{InCate2} becomes comparable to that of Π_{InCate3}. Moreover, Π_{InCate3} facilitates the use of activation function parity, reducing the cost of evaluating the non-linear $f(x)$ in Equation 1. Therefore, Π_{InCate3} is more suitable for evaluating functions in Category 3.

Algorithm 4: Interval test for Category 3, Π^l_{InCate3}:

Input: P_0 and P_1 hold $[\![x]\!]^l$, and public $\beta = -\alpha = 2^h$.
Output: P_0 and P_1 output $[\![b]\!]^B$, $[\![b^*]\!]^B$ and $[\![b^\#]\!]^B$ defined in Equation 5.

1. P_0 & P_1 invoke $\mathcal{F}^h_{\mathsf{TR}}([\![x]\!]^l)$ to learn $[\![t]\!]^{l-h}$.
2. P_0 & P_1 invoke $\mathcal{F}^{l-h}_{\mathsf{DReLU\&negone}}([\![t]\!]^{l-h})$ to learn $[\![b]\!]^B$ and $[\![b^*]\!]^B$.
3. P_0 & P_1 invoke $\mathcal{F}^{l-h}_{\mathsf{Zero}}([\![t]\!]^{l-h})$ to learn $[\![b^\#]\!]^B$.
4. P_0 & P_1 output $[\![b]\!]^B$, $[\![b^*]\!]^B$ and $[\![b^\#]\!]^B$.

4 Evaluating Activation Functions with Dynamic Precision

Section 3 introduces new interval test protocols and the methods to compute $\mathsf{F}(x)$ from $f(x)$, $\tilde{L}(x)$ and $\hat{L}(x)$. Consequently, the remaining challenge is evaluating non-linear $f(x)$. In this section, we introduce the dynamic precision method to evaluate $f(x)$ and provide specific algorithms for evaluating activation functions.

4.1 Properties of Common Activation Functions

We start by examining the key properties of common activation functions, including their non-linear intervals, expressions of the linear functions $\hat{L}$ and $\tilde{L}$, parity, and their classification, all of which are essential for efficient evaluation. Additionally, we examine the range of slope and intercept in the linear approximation method. Specifically, we approximate $f(x)$ using a linear function $l_i = a_i x + d_i$ within each interval $[\gamma_i, \gamma_{i+1})$ for $i \in [2^s]$. Intuitively, for monotonic $f(x)$, the optimal approximation satisfies $|a_i| \leq \max\{f'(\gamma_i), f'(\gamma_{i+1})\}$, where f' is the derivative of f. To formalize this, we introduce two functions $\mathsf{AF}'(x)$ and $D_{\mathsf{AF}}(x)$, where $\mathsf{AF}'(x)$ is the derivative of $\mathsf{AF}(x)$, and the function $D_{\mathsf{AF}}(x)$ is defined as $D_{\mathsf{AF}}(x) = \mathsf{AF}(x) - \mathsf{AF}'(x) \cdot x$. Then the range of a_i and d_i can be estimated by $\mathsf{AF}'(x)$ and $D_{\mathsf{AF}}(x)$, respectively. These properties are summarized in Table 2. It is noteworthy that we can compute $g(x)$ to evaluate $\mathsf{GELU}(x)$(see Section 2.5 for

details), which is an even function and can be classified under Category 3.

4.2 Evaluating a Non-linear Function $f(x)$

We employ the linear approximation method described in Equation 2 to evaluate $f(x)$ using piecewise linear functions $l_i = a_i x + d_i$ for $i \in [n]$. In this approach, the non-linear interval $[\alpha, \beta)$ of size 2^h is divided into $n = 2^s$ sub-intervals. The evaluation begins with modulus and truncation operations to compute $I = (x \mod 2^h) \gg (h-s)$, which is the index of the sub-interval x falls into. Then the slope $a = T_a[I]$ and intercept $d = T_d[I]$ are retrieved from pre-constructed lookup tables T_a and T_d. Finally, a multiplication protocol is invoked to evaluate $f(x) \approx ax + d$.

The overhead of this method primarily lies in the LUT and multiplication protocols, which depend on the bitwidths of the slope, intercept and input x. The accuracy is influenced by s and the values stored in T_a and T_b. Prior works [11, 15] typically set the bitwidth and precisions of a and d equal to those of x, resulting in large lookup tables and significant overhead for multiplication protocol. Moreover, the values in lookup tables may fail to achieve high approximation accuracy. To address these issues, we propose a dynamic precision method to reduce the bitwidths of a and d, and construct high-accuracy lookup tables, enabling high accuracy with reduced overhead.

4.2.1 Evaluating $f(x)$ with low bitwidth and precision

Let the bitwidth and precision of input x be l and f, respectively, while the bitwidths and precisions of the slope and intercept are denoted as l_a, f_a, l_d and f_d. The linear approximation method approximates $f(x)$ using $l_i = a_i x + d_i$ within the interval $[\gamma_i, \gamma_{i+1})$. From Section 4.1 and Table 2, we can constrain a_i and d_i to a relatively small range, namely, $(-1, 1)$. Consequently, only the fractional parts of a_i and d_i need to be represented, allowing their bitwidths to be reduced to $l_a = f_a + 1$ and $l_d = f_d + 1$ bits, respectively.

To further reduce the bitwidths, we apply dynamic precisions method. Specifically, we let the precisions of a_i and d_i be smaller than f, resulting in bitwidths reduced to $l_a = f_a + 1$ and $l_d = f_d + 1$. Given the parameters f_a and f_d and pre-constructed lookup tables T_a and T_d, we propose Π^l_{nlinear} in Algorithm 5 to evaluate $f(x)$. In this algorithm, $f(x)$ is evaluated over the non-linear interval $[-2^h, 0)$ for functions in Category 2, and $[0, 2^h)$ for functions in Category 3, leveraging the parity of $f(x)$. When using dynamic precisions, additional operations such as truncation, modulus, multiplication by constants and extension are required to align the bitwidths and precisions between $a_i \cdot x$ and d_i before performing the addition, while these operations only take small overhead. It should be noted that the signs of the inputs of Π_{SMul}, Π_{AppTR} and Π_{SExt} are known in public, therefore these protocols can be implemented more efficiently. Now, the remaining challenges include determining appropriate parameters for f_a and f_d and constructing high accuracy lookup tables.

Algorithm 5: Evaluating the non-linear function $f(x)$, Π^l_{nlinear}:

Input: P_0 and P_1 hold $[\![x]\!]^l$ with f-bit precision, a public function $f(x)$ with non-linear interval $[-2^h, 0)$ or $[0, 2^h)$. the precisions f_a and f_d, s and and two lookup tables T_a and T_d are given in advance.

Output: P_0 and P_1 output $[\![z]\!]^l$ where z is the approximation of $f(x)$ as Equation 2.

1. Let $[\![x']\!]^l = [\![x]\!]^l$.
2. **if** *non-linear interval is* $[-2^h, 0)$ **then**
3. $\quad [\![x']\!]^l = [\![x]\!]^l + 2^h$.
4. **end**
5. For $i \in \{0,1\}$, P_i computes $[\![x]\!]^h_i = [\![x']\!]^l_i \mod 2^h$, then they invoke $\mathcal{F}^{h-s}_{\mathsf{TR}}([\![x]\!]^h)$ to learn $[\![I]\!]^s$.
6. P_0 & P_1 invoke $\mathcal{F}_{\mathsf{LUT}}(T_a, [\![I]\!]^s)$ and $\mathcal{F}_{\mathsf{LUT}}(T_d, [\![I]\!]^s)$ to learn $[\![a]\!]^{l_a}$ and $[\![d]\!]^{l_d}$.
   `// Compute z = ax + d`
7. P_0 & P_1 invoke $\mathcal{F}^{l_a,l}_{\mathsf{SMul}}([\![a]\!]^{l_a}, [\![x]\!]^l)$ to learn $[\![y]\!]^{l+l_a}$.
8. P_0 & P_1 invoke $\mathcal{F}^{f_a}_{\mathsf{AppTR}}([\![y]\!]^{l+l_a})$ to learn $[\![y]\!]^{l+l_a-f_a}$.
9. For $i \in \{0,1\}$, P_i sets $[\![y]\!]^l_i = [\![y]\!]^{l+l_a-f_a}_i \mod 2^l$.
10. For $i \in \{0,1\}$, P_i sets $[\![d]\!]^{l_d+f-f_d}_i = [\![d]\!]^{l_d}_i \cdot 2^{f-f_d} \mod 2^{l_d+f-f_d}$.
11. P_0 & P_1 invoke $\mathcal{F}^{l_d+f-f_d,l}_{\mathsf{SExt}}([\![d]\!]^{l_d+f-f_d})$ to learn $[\![d]\!]^l$.
12. P_0 & P_1 compute $[\![z]\!]^l = [\![y]\!]^l + [\![d]\!]^l$.

4.2.2 Setting precisions and constructing lookup tables

Prior works directly set the precisions of slope and intercept to match the precision of the input x, thereby introducing significant overhead. In this work, we take both the overhead and accuracy into consideration, and propose a dynamic precision method to balance overhead and accuracy. Our approach involves directly searching for optimal parameters f_a and f_d, which minimize overhead while ensuring that the error remains within a given bound. Specifically, we define an overhead function $C(f_a, f_d)$ to describe the communication cost of Π_{nlinear}. Additionally, we introduce an error function $\mathsf{MinError}^{[\gamma_i,\gamma_{i+1})}(f_a, f_d)$, which measures the minimum error between the linear function $l_i = a_i x + d_i$ and the target function $f(x)$ over the interval $[\gamma_i, \gamma_{i+1})$. This minimum is computed by optimizing over the precisions f_a and f_d of the coefficients a_i and d_i, respectively. Given a specified error threshold ε, we formulate an optimization model where the objective is to minimize $C(f_a, f_d)$, subject to the constraint that the minimum error $\mathsf{MinError}^{[\gamma_i,\gamma_{i+1})}(f_a, f_d)$ for each $i \in [2^s]$ does not exceed ε. By solving this model, we obtain the optimal pre-

Table 2: Some properties of common activation functions.

Activation Function	non-linear interval	$\hat{L}$	$\tilde{L}$	AF'	D_{AF}	Parity	Category
ReLU	-	0	x	-	-	no	1
ELU ($\alpha=1$)	$[-8,0)$	-1	x	$(0,1]$	$(-1,0]$	no	2
Tanh	$[-4,4)$	1	-1	$(0,1]$	$(0,1)$	odd	3
Sigmoid (shifted)	$[-8,8)$	-0.5	0.5	$(0,0.25]$	$(-0.5,0.5)$	odd	3
$g(x)$(GELU)	$[-4,4)$	$\frac{x}{2}$	$-\frac{x}{2}$	$\approx(-0.63,0.63)$	$\approx(-0.29,0]$	even	3
Tanh$'$	$[-4,4)$	0	0	$(0,1]$	$(-1,1)$	even	4
Sigmoid$'$ (shifted)	$[-8,8)$	0	0	$(0,0.25]$	$(0,1)$	even	4

cisions f_a and f_d corresponding to the given error bound ε. Additionally, the lookup tables are constructed using the intermediate variables generated during the optimization process. We give the details of this procedure in the following.

Overhead function. In Π^l_{nlinear}, the parameters h, f and s are predetermined in advance and can be treated as constants. Thus, the total communication cost of this protocol depends on the bitwidths l_a and l_d, which is approximately as:

$$\hat{C}(l_a,l_d) = (2\lambda+2^s+2l+l_a+4)l_a+l_d\cdot 2^s+15\lambda + 3l-f+(\lambda+14)(h-s).$$

Since $l_a=f_a+1$ and $l_d=f_d+1$, we can define the overhead function $C(f_a,f_d)$ as:

$$C(f_a,f_d)=\hat{C}(f_a+1,f_d+1). \tag{8}$$

Error function. The error function is defined in plaintext mode, which is equivalent to its secret-sharing counterpart. An l-bit real number x with precision f is represented as $x=q\cdot 2^{-f}$, where $q\in\mathbb{Z}$ and $-2^{l-f-1}\le q<2^{l-f-1}$. Similarly, let $\gamma_i=k_i\cdot 2^{-f}$ and $\gamma_{i+1}=k_{i+1}\cdot 2^{-f}$, and then $x\in[\gamma_i,\gamma_{i+1})$ is represented as $x=q\cdot 2^{-f}$ where $k_i\le q<k_{i+1}$. For a_i, d_i $\in(-1,1)$ with precision f_a and f_d, we write $a_i=m\cdot 2^{-f_a}$ and $d_i=n\cdot 2^{-f_d}$, where $m,n\in\mathbb{Z}$, $-2^{f_a}<m<2^{f_a}$ and $-2^{f_d}<n<2^{f_d}$. Using the maximum ULP (Unit in the Last Place) error measure [9], the error between $l_i=a_ix+d_i$ and $f(x)$ in interval $[\gamma_i,\gamma_{i+1})$ is:

$$\begin{aligned}&\mathsf{Error}^{[\gamma_i,\gamma_{i+1})}(f_a,f_d,a_i,d_i)\\ =&\max_{x\in[\gamma_i,\gamma_{i+1})}\{\frac{|a_ix+d_i-f(x)|}{2^{-f}}\}\\ =&\max_{k_i\le q<k_{i+1}}\{\frac{|m_i\cdot 2^{-f_a}\cdot q\cdot 2^{-f}+n_i\cdot 2^{-f_d}-f(q\cdot 2^{-f})|}{2^{-f}}\}.\end{aligned}$$

The minimum error for given f_a and f_d in interval $[\gamma_i,\gamma_{i+1})$ is defined as:

$$\begin{aligned}&\mathsf{MinError}^{[\gamma_i,\gamma_{i+1})}(f_a,f_d)\\ =&\min_{\substack{-2^{f_a}<m_i<2^{f_a},\\ -2^{f_d}<n_i<2^{f_d}}}\mathsf{Error}^{[\gamma_i,\gamma_{i+1})}(f_a,f_d,m_i\cdot 2^{f_a},n_i\cdot 2^{f_d}).\end{aligned}$$

It is important to note that (m_i,n_i) can be different across intervals, while the same f_a and f_d are applied to all subintervals.

Optimization model. Based on the overhead and error functions, the optimization model is formulated as:

$$\begin{aligned}\min_{f_a,f_d\in\mathbb{Z}^+}\quad & C(f_a,f_d)\\ \text{s.t.}\quad & \mathsf{MinError}^{[\gamma_i,\gamma_{i+1})}(f_a,f_d)\le\varepsilon\\ & \text{for } i=0,1,\dots,2^s-1.\end{aligned}$$

This model can be solved using the Gurobi optimization solver [2], yielding the optimal f_a and f_d for minimal overhead. The intermediate variables m_i and n_i for $i\in[2^s]$ are used to construct lookup tables $T_a[i]=m_i\cdot 2^{-f_a}$ and $T_d[i]=n_i\cdot 2^{-f_d}$. These parameters enable high accuracy with low overhead.

4.3 Applying to Activation Functions

Combining all the proposed techniques, activation functions can be evaluated following the workflow illustrated in Figure 1. For a given activation function AF, its properties are first determined as Table 2. Subsequently, the dynamic precision method is employed to obtain f_a, f_d, and to construct the lookup tables. These operations are executed offline and in plaintext, requiring only a one-time setup for each activation function. In the online phase, the interval test protocol Π_{InCate2} or Π_{InCate3}, and the evaluation protocol Π_{nlinear} are invoked. The approximation $\mathsf{F}(x)$ of $\mathsf{AF}(x)$ is then computed from $f(x)$, $\hat{L}(x)$ and $\tilde{L}(x)$, as described in Equation 4 or Equation 6. In summary, evaluating $\mathsf{F}(x)$ involves invoking protocols including Π_{TR} (to extract the middle s bits), two instances of Π_{LUT}, Π_{Mul}, an interval test protocol and several instances of Π_{MUX}. Below, we detail the evaluation algorithm for specific activation functions.

Evaluating activation functions in Category 2. For activation functions in this category, Π^l_{InCate2} is invoked to determine the interval in which x falls. Then Π_{nlinear} is invoked to evaluate $f(x)$. Finally, $\mathsf{F}(x)$ is computed using Equation 3. The details are outlined in Algorithm 6. It is worth noting that Π^l_{InCate2} and Π_{nlinear} can be performed in parallel, enabling further optimization.

Algorithm 6: Evaluating activation functions in Category 2, Π_{FCate2}:

Input: P_0 and P_1 hold $[\![x]\!]^l$ with f-bit precision, and an activation function $\mathsf{AF}(x)$ with non-linear interval $[-2^h, 0)$.

Output: P_0 and P_1 output $[\![res]\!]^l$ such that $res \approx \mathsf{AF}(x)$.

1 **Preprocess (in plaintext, only once):**
2 Approximate $\mathsf{AF}(x)$ as Equation 1, and set the value of s depending on the size of non-linear interval and required accuracy.
3 Based on $\mathsf{AF}(x)$, construct and solve the optimization model using the method in Section 4.2 to get f_a, f_d, and generate the lookup tables T_a and T_b.
4 **Online:**
5 P_0 & P_1 compute $[\![L_l]\!]^l$ and $[\![L_r]\!]^l$ based on linear function $\hat{L}$ and $\tilde{L}$.
6 P_0 & P_1 invoke the following functionalities:
7 $\mathcal{F}_{\mathsf{InCate2}}([\![x]\!]^l)$ to learn $[\![b]\!]^B$ and $[\![b^*]\!]^B$.
8 $\mathcal{F}_{\mathsf{nlinear}}([\![x]\!]^l, s, f_a, f_d, T_a, T_b)$ to learn $[\![z]\!]^l$.
9 $\mathcal{F}^l_{\mathsf{MUX}}([\![z]\!]^l - [\![L_r]\!]^l, [\![b^*]\!]^B)$ to learn $[\![tA]\!]^l$.
10 $\mathcal{F}^l_{\mathsf{MUX}}([\![L_r]\!]^l - [\![L_l]\!]^l, [\![b]\!]^B)$ to learn $[\![tB]\!]^l$.
11 P_0 & P_1 compute $[\![res]\!]^l = [\![L_r]\!]^l + [\![tA]\!]^l + [\![tB]\!]^l$.
12 P_0 & P_1 output $[\![res]\!]^l$.

Activation functions in Category 3. To reduce overhead, parity is leveraged in evaluating functions in Category 3. First, $\mathcal{F}_{\mathsf{InCate3}}$ is invoked to identify the interval in which x falls. The absolute value $|x|$ is then computed, focusing on the positive non-linear interval $[0, \beta)$ for evaluating $f(|x|)$. Finally, $\mathsf{F}(|x|)$ and $\mathsf{F}(x)$ are computed using Equation 6.

However, an issue arises when $x = \alpha = -2^h$. In this case, $\mathcal{F}_{\mathsf{InCate3}}$ outputs $(b, b^*, b^\#) = (0,1,0)$, and $\mathsf{F}(|-2^h|)$ should be evaluated as $a_{n-1}x + d_{n-1}$. While when retrieving (a,d) from lookup tables, the index I is computed as $I = (|x| \bmod 2^h) \gg (h-s) = 0$. This results in the slope and intercept being a_0 and d_0, leading to $\mathsf{F}(|-2^h|)$ being calculated as $a_0x + d_0$. For certain activation functions such as Tanh and $\mathsf{Sigmoid}$, $(a_0, d_0) \approx (a_{n-1}, d_{n-1}) \approx (0,0)$, so this error has negligible impact on the correctness. But for $g(x)$, which is used to evaluate GELU, the discrepancy between $a_0 \cdot 2^h + d_0$ and $a_{n-1} \cdot 2^h + d_{n-1}$ is significant, resulting in substantial errors. It is worth noting that although the index I is also 0 for $x = 2^h$, this error does not occur as $\mathcal{F}_{\mathsf{InCate3}}$ outputs $(b, b^*, b^\#) = (1,0,0)$, and $\mathsf{F}(|2^h|)$ is evaluated by $\hat{L}(|2^h|)$. We address this issue by slightly modifying the absolute function to $\mathsf{mABS}(x) = |x| - \mathsf{MSB}(x)$, which outputs $|x| - 1$ for negative inputs. Then, for $x = -2^h$, this modification ensures $\mathsf{mABS}(x) = 2^h - 1$, allowing the correct (a_{n-1}, d_{n-1}) to be retrieved from the lookup tables. For other values of x, this approach can also provide approximately correct slopes and intercepts. The detailed evaluation process for activation functions in Category 3 is summarized in Algorithm 7. For evaluation on GELU, we first evaluate $g(x)$ and compute $\frac{x}{2}$ by invoking a truncate 1-bit protocol, then $\mathsf{GELU} = g(x) + \frac{x}{2}$.

Algorithm 7: Evaluating activation functions in Category 3, Π_{FCate3}:

Input: P_0 and P_1 hold $[\![x]\!]^l$ with f-bit precision, and an odd or even function $\mathsf{AF}(x)$ with non-linear interval $[-2^h, 2^h)$.

Output: P_0 and P_1 output $[\![res]\!]^l$ such that $res \approx \mathsf{AF}(x)$.

1 **Preprocess (in plaintext, only once):**
2 Approximate $\mathsf{AF}(x)$ as Equation 1, and set the value of s depending on the size of non-linear interval and required accuracy.
3 Let $p = 1$ if $\mathsf{F}(x)$ is odd function, and $p = 0$ otherwise.
4 Based on $\mathsf{AF}(x)$, construct and solve the optimization model using the method in Section 4.2 to get f_a, f_d, and generate the lookup tables T_a and T_b.
5 Let $\delta = 0$ if $(T_a[0], T_d[0]) \approx (T_a[2^s-1], T_a[2^s-1])$, else $\delta = 1$.
6 **Online:**
7 P_0 & P_1 compute $[\![L_r]\!]^l$ based on linear function $\hat{L}$.
8 P_0 & P_1 invoke the following functionalities:
9 $\mathcal{F}_{\mathsf{InCate3}}([\![x]\!]^l)$ to learn $[\![b]\!]^B$, $[\![b^*]\!]^B$ and $[\![b^\#]\!]^B$.
10 $\mathcal{F}^l_{\mathsf{SS}}([\![x]\!]^l, -[\![x]\!]^l - \delta, [\![b]\!]^B)$ to learn $[\![x_{abs}]\!]^l$.
11 $\mathcal{F}_{\mathsf{nlinear}}([\![x_{abs}]\!]^l, s, f_a, f_d, T_a, T_b)$ to learn $[\![z]\!]^l$.
12 $\mathcal{F}^l_{\mathsf{SS}}([\![z]\!]^l, [\![L_r]\!]^l, [\![b^*]\!]^B \oplus [\![b^\#]\!]^B)$ to learn $[\![F_{abs}]\!]^l$.
13 $\mathcal{F}^l_{\mathsf{SS}}([\![F_{abs}]\!]^l, (-1)^p \cdot [\![F_{abs}]\!]^l, [\![b]\!]^B)$ to learn $[\![res]\!]^l$.
14 P_0 & P_1 output $[\![res]\!]^l$.

Activation functions in Category 4. The key distinction of functions in Category 4 is that their linear components are identical, namely, $\hat{L} = \tilde{L}$. As a result, it is sufficient to identify the non-linear interval without differentiating between $\hat{L}$ and $\tilde{L}$. This allows the use of the more efficient interval test protocol $\Pi^l_{\mathsf{InSingle}}$, instead of Π_{InCate2} or Π_{InCate3}. Furthermore, evaluating $\mathsf{F}(x)$ is simplified, as demonstrated in Equation 7. Thus, functions in this category can be evaluated with higher efficiency.

4.4 Complexity Analysis

The communication costs of Algorithm 6 and Algorithm 7 are summarized in Table 3, assuming that the functions in Category 3 are odd functions. The parameter s is usually set to 6 or 7 and can be regarded as a constant. The results reveal that, aside from the bitwidth l, the total communication is primarily influenced by l_a. Consequently, reducing the bitwidth (or precision) of a significantly decreases the overall communication costs of the protocol.

Table 3: Communication of evaluating activation functions in Category 2 and Category 3.

	Category 2.	Category 3
Inter.	$\lambda l + 14l$	$\frac{7l-3h}{4}\lambda + 14l - 4h$
$f(x)$	$C(l_a, l_d)$	$C(l_a, l_d)$
Others.	$4(\lambda + 2l)$	$6(\lambda + 2l)$
Total.	$\lambda l + (23 + 2l_a)\lambda + (26 + 2l_a)l + 2^s(l_a + l_d) + l_a^2 + 3l_a + 112$	$\frac{7}{4}\lambda l + (25 + 2l_a - \frac{3}{4}h)\lambda + (30 + 2l_a)l + 2^s(l_a + l_d) + l_a^2 + 3l_a - 4h + 112$

Table 4: Comparison of our interval test protocols with prior approach, with bitwidth $l = 21$. Runtime represents the total execution time (in seconds) for 2^{20} runs of the protocols, while the communication denotes the data exchanged (in bits) per execution.

Protocol.	Runtime		Comm. (per instance)
	LAN	WAN	
Two Π_{DReLU}	11.84	57.33	4288
Π^l_{InSingle}	6.00	34.41	2504
	1.97×	**1.66×**	**1.71×**
Π^l_{InCate2}	6.47	37.96	2800
	1.82×	**1.51×**	**1.53×**
Π^l_{InCate3}	8.30	47.40	3488
	1.42×	**1.20×**	**1.22×**

5 Experiment

Experimental setup. We conducted our experiments on a server equipped with Intel Xeon processors operating at 2.3 GHz, featuring 64 cores and 128 GB of memory. To simulate various network conditions, we used Linux Traffic Control (tc), emulating a LAN environment with 1 Gbps bandwidth and 0.8 ms RTT latency, and a WAN environment with 100 Mbps bandwidth and 80 ms RTT latency. Except for the experiment in subsection 5.3.3, all experiments were implemented on top of the SCI library [1], utilizing IKNP-style OT protocols. This ensure fairness and consistency in benchmarking, as all prior works used for comparison also adopted IKNP-style OT. The implementations are publicly available at https://github.com/geralt-tian/SEAF. Additionally, following the Cheetah framework [12], we implemented SEAF using VOLE-style OT protocol (in subsection 5.3.3). The implementations are available at https://github.com/geralt-tian/SEAF-VOLE-AF.

5.1 Interval Test Protocol

We implemented our new interval test protocols and compared their performance with prior approach that invokes two DReLU protocols in Bolt [18]. The experimental results are presented in Table 4, where the input bitwidth is set to $l = 21$. The experimental results indicate that the overhead of our single test protocol Π_{InSingle} is comparable to that of a single DReLU protocol, reducing the communication costs by 41.6%, and improving the runtime by 1.66× to 1.97×. The Π_{InCate2} incurs slightly higher overhead than Π_{InSingle}, as it replaces Π_{Zero} with $\Pi_{\text{DReLU\&negone}}$. Despite this, it achieves a 1.53× reduction in communication costs and a runtime improvement of 1.51× to 1.82×. The Π_{InCate3} extends Π_{InCate2} by invoking an additional Π_{Zero}, therefore, with lower efficiency. However, it still outperforms the prior approach, achieving 1.2× to 1.42× improvement in runtime.

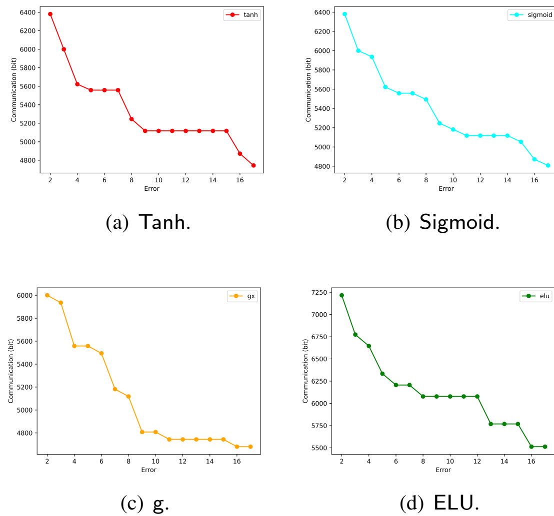
5.2 Evaluating non-linear $f(x)$

The key for evaluating $f(x)$ involves determining the precisions of slope and intercept and generating lookup tables. In this subsection, we follow the method in Section 4.2 and conduct experiments on several real-world activation functions to analyze the relationship between the overhead and the approximation accuracy. For input x, we set its bitwidth to $l = 21$, and precision to $f = 12$. The non-linear intervals for these functions are provided in Table 2. For a given maximum ULP error $\varepsilon \in \{2, 3, ..., 18\}$, we construct and solve the optimization models for several activation functions as described in Section 4.2 to obtain the minimum overhead. The results are presented in Figure 5.

In Figure 5, the error and overhead exhibit an approximately inverse relationship, aligning with the intuition that higher accuracy approximations necessitate greater overhead. Notably, the curve in Figure 5 is steep for small errors but gentler for larger errors. This indicates that the overhead is significantly influenced by error reduction when high accuracy is required, whereas the impact diminishes for lower accuracy. Take Tanh as an example, reducing the error from 4 to 2 increases communication by 758 bits (from 5622 to 6380), whereas reducing the error from 16 to 8 only increases communication by 374 bits (from 5246 to 4872). Furthermore, we observed that in some cases, communication remains unchanged even when the error bound is relaxed. This suggests that increasing the error may not always incur additional overhead, as the minimum communication for certain error ranges can be identical. For example, in Figure 5(a), the communication for evaluating Tanh remains constant at 5118 bits for error bounds ranging from 9 to 15. When the error bound is $\varepsilon = 15$, the minimum communication is 5118 bits, corresponding to precision values $f_a = 4$ and $f_d = 9$. However, under these parameters, the minimum achievable error is 9, implying that the actual error may be even lower.

The communications in Figure 5 are calculated using Equa-

tion 8, representing theoretical estimates. When applied to the complete activation functions, the error may increase slightly due to the use of efficient but approximate sub-protocols, such as Π_{AppTR} in Π_{nlinear}. In the following subsections, we present comprehensive experiments to demonstrate the actual error incurred when evaluating real-world activation functions.



(a) Tanh. (b) Sigmoid. (c) g. (d) ELU.

Figure 5: The minimum communication (theoretical) of evaluating non-linear $f(x)$ for given error $\varepsilon \in \{2,3,\ldots,18\}$ for functions Tanh, Sigmoid, g and ELU, where $l = 21$.

5.3 Evaluations on Activation functions

Following the workflow outlined in Figure 1, we implemented SEAF to evaluate real-world activation functions. Initially, SEAF was developed using IKNP-style OT and compared with prior works such as SirNN [20], Iron [10], and Bolt [18], which also adopt IKNP-style OT. This ensures a fair and consistent comparison. Additionally, drawing inspiration from recent advancements like Cheetah [12] and BumbleBee [16], we implemented SEAF using VOLE-style OT [4] as the underlying OT protocol.

5.3.1 Evaluation on SEAF

The experimental results of SEAF with IKNP-style OT are presented in Table 7 in Appendix C. In these experiments, the bitwidth and precision of the input are set to $l = 21$ and $f = 12$, respectively. The non-linear interval is divided into 2^s segments, where $s = 6$ for Tanh, Sigmoid, and GELU, and $s = 7$ for ELU, as the steeper curve of ELU necessitates more segments to achieve smaller errors. Both the average ULP error and maximum ULP error are computed within the non-linear interval. The second column of Table 7 presents results obtained using general bitwidths $l_a = l_d = l$ and fixed precision $f_a = f_d = f$, while we also use the method in Section 4.2.2 to generate lookup tables. Compared to the fixed precision method, our dynamic precision technique reduces communication costs by approximately 29% to 48%. Furthermore, the overhead is primarily influenced by f_a, as it determines the cost of the multiplication $a \cdot x$. While f_d has a minor impact on overhead, it significantly affects accuracy. To optimize performance while maintaining high accuracy, f_a should be set small while f_d can be set to a larger value.

One notable advantage of SEAF is its adjustable trade-off between overhead and accuracy. For applications requiring low accuracy, lower-precision parameters can be employed to reduce overhead. Table 7 serves as a reference for determining the relationship between precision parameters and fitting errors. Given a maximum acceptable error (either maximum ULP or average ULP), one can consult Table 7 to identify the optimal precision values f_a and f_d that minimize overhead.

5.3.2 Comparison with prior works

Based on Table 7, we roughly set three version of SEAF: SEAF^1, SEAF^2 and SEAF^3, corresponding to high, medium, and low accuracy. These variants were compared against SirNN [20] and Bolt [18], with the results summarized in Table 5. For Tanh and Sigmoid, SEAF reduces communication costs by 74% to 79%, and achieves $3.5\times$ to $5.9\times$ improvement in runtime compared to SirNN. Moreover, the high accuracy SEAF^1 demonstrates comparable or lower ULP errors relative to SirNN. The slightly higher errors observed in SEAF^2 and SEAF^3 are attributed to their use of lower precision parameters f_a and f_d, which enhance efficiency. For GELU, SEAF significantly outperforms Bolt [18], reducing the communication costs by nearly 61% to 69%, and improves the runtime by $1.8\times$ to $3.4\times$. In terms of accuracy, both SEAF^1 and SEAF^2 exhibit substantial error reductions, encompassing both maximum and average ULP errors. Even the low accuracy SEAF^3 surpasses Bolt in average ULP performance, with only a slight increase in maximum ULP errors (from 14 to 17).

5.3.3 Implementation with VOLE-style OT

Compared to IKNP-style OT, VOLE-style OT introduces greater computational overhead but achieves lower communication costs. Therefore, some frameworks such as Cheetah [12] and BumbleBee [16] adopt VOLE-style OT as the foundational component to minimize the communication. In this work, we follow the Cheetah framework and implemented SEAF using VOLE-style OT, with the experimental result presented in Table 8. Compared to the IKNP-style OT implementation (shown in Table 7), the VOLE version achieves reduced communication overhead. This results in a runtime improvement of approximately 2 to 3 times under a WAN environment. However, in LAN environment, while the ad-

Table 5: Comparison of SEAF with prior works on evaluating Tanh, Sigmoid and GELU. The bitwidth and precision of input x are set as $l = 21$ and $f = 12$. The bitwidths of slope and intercept are set as $(l_a, l_d) = (f_a + 1, f_d + 1)$. The results of SEAF are listed with high, medium and low accuracy, corresponding to SEAF^1, SEAF^2 and SEAF^3. The communication and timing are accumulated for 2^{20} runs of the protocols.

	Protocol.	(f_a, f_d)	Time. (s)		Comm. (GB)	avg ULP	max ULP
			LAN	WAN			
Tanh	SirNN [20]	-	85.204	460.037	5.439	1.34	4
	SEAF^1	$(6,12)$	19.137	115.553	1.420	0.82	3
	SEAF^2	$(4,10)$	17.886	105.950	1.242	1.67	9
	SEAF^3	$(3,8)$	14.462	100.562	1.151	5.53	18
Sigmoid	SirNN [20]	-	68.703	466.558	5.509	0.95	3
	SEAF^1	$(8,12)$	19.68	125.420	1.438	1.07	3
	SEAF^2	$(5,10)$	18.545	111.391	1.373	1.74	7
	SEAF^3	$(3,11)$	18.448	104.077	1.280	2.73	17
GELU	Bolt [18]	-	34.517	324.331	3.827	5.91	14
	SEAF^1	$(7,12)$	19.224	119.898	1.477	1.09	3
	SEAF^2	$(4,10)$	18.189	106.572	1.248	1.53	7
	SEAF^3	$(3,8)$	15.306	94.961	1.184	4.19	17

vantage of reduced communication persists, the substantial computational overhead introduced by VOLE-OT has a more significant impact on efficiency, leading to slightly lower overall performance compared to the IKNP-based implementation.

5.4 End-to-End Performance

We conducted end-to-end secure inference experiments on BERT [6] to demonstrate the improvements introduced by SEAF. Specifically, we integrate SEAF into the frameworks of Iron [10] and Bolt (with Word Elimination) by replacing their GELU implementations with our optimized version. For our GELU, we set the bitwidths and precisions of slope and intercept as $(l_a, l_d) = (4, 11)$ and $(f_a, f_d) = (3, 10)$. In this precision, the maximum ULP and average ULP error are 7 and 1.53, respectively, which are smaller than those of Bolt's GELU. Therefore, SEAF achieved similar accuracy than Bolt on secure inference task. The experiments were conducted under WAN settings, as GELU contributes only minimally to the total overhead in LAN settings for Bolt. The experiment results are summarized in Table 6 and Figure 6. Compared to Iron's GELU, our optimized GELU reduces communication costs by 90% and the number of communication rounds by 80 %. Consequently, the total communication and communication rounds for secure inference are reduced by approximately 30% and 18%, respectively, and the total runtime is improved by nearly 1.39×. Furthermore, the proportion of runtime attributable to GELU decreases from 31.3% to 4.3%. When compared to Bolt, our approach reduces end-to-end communication by over 20%, and improves the total runtime by 1.16×. Moreover, our improved GELU accounts for only 9.8% of the total runtime, compared to 22.5% in the original Bolt implementation. As a result, GELU is no longer a primary overhead in Iron+SEAF and Bolt+SEAF frameworks.

Table 6: Communication and rounds comparing Iron, Bolt, with SEAF. The communication is measured in GB.

	GELU		Total	
	Comm.	Round	Comm.	Round
Iron	90.16	4152	276.25	18106
SEAF + Iron	8.61	864	194.49	14818
	10.47×	**4.81**×	**1.42**×	**1.22**×
Bolt	8.73	1440	25.71	14452
SEAF + Bolt	3.15	888	20.13	13900
	2.77×	**1.62**×	**1.28**×	**1.04**×

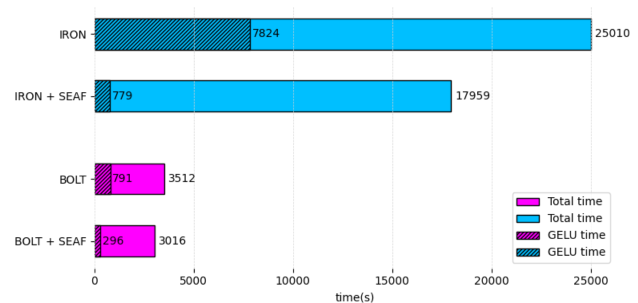


Figure 6: Comparison of runtime (in seconds) among Iron, Bolt and SEAF for the end-to-end inference of BERT.

6 Conclusion

This work presents SEAF, a novel framework designed for the efficient evaluation of common non-linear activation functions. SEAF integrates two core techniques: Trun-Eq based interval test protocols and linear approximation with dynamic precision. These optimizations deliver significant performance improvements for real-world activation functions, enhancing their efficiency by several times. Furthermore, by leveraging these optimized activation functions, SEAF significantly improves the overall efficiency of end-to-end secure inference.

The proposed techniques have the potential applications beyond the scope of this work. Interval test is a common challenge in many scenarios, where our optimizations can be effectively utilized. Furthermore, the dynamic precision approximation method may be particularly well-suited for evaluating other general non-linear functions, such as the normal distribution function. We believe that SEAF can achieve strong performance in these applications as well.

Acknowledgments

We thank the anonymous shepherd and reviewers for their valuable feedback. We also thank Chunzao Huang and Liqiang Peng for their help and suggestions. This work was supported in part by Shenzhen Colleges and Universities Stable Support Program (Grant No. GXWD20231129135251001), National Natural Science Foundation of China (Grant No. 62301190), Shenzhen Fundamental Research Program (Grant No. JCYJ20241202124023031) and Zhejiang Provincial Natural Science Foundation of China (Grant No. LQN25F020030).

Ethics Considerations and Compliance

All authors of this paper collectively affirm the following:

1. We confirm that we have thoroughly reviewed the ethics considerations outlined in the conference call for papers, the detailed submission instructions, and the accompanying guidelines for ethics documentation.

2. We attest that our research complies with all ethical guidelines and open science policy, and no ethical concerns are relevant to this study. As the research did not involve human participants, data privacy concerns, or other sensitive issues, the need for informed consent does not apply. We believe our team's next-step plans (e.g., after publication) are ethical.

3. We affirm that our research adheres to all applicable ethical standards and the principles of the Open Science Policy. This study does not involve human participants, data privacy issues, or other sensitive matters; therefore, obtaining informed consent is not applicable. Furthermore, we confirm that our team's planned future work (e.g., post-publication) complies with ethical principles.

Compliance with the Open Science Policy

We fully endorse the principles of the Open Science Policy. Our research artifacts have been uploaded to GitHub repository. These artifacts will be made publicly available with the final version of the paper, ensuring unrestricted access for the scientific community to review, validate, and build upon our work.

References

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

[2] Gurobi Optimization. Gurobi Optimizer Reference Manual. 2019. https://www.gurobi.com/.

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

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

[5] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

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

[7] Ye Dong, Wen-jie Lu, Yancheng Zheng, Haoqi Wu, Derun Zhao, Jin Tan, Zhicong Huang, Cheng Hong, Tao

Wei, and Wenguang Chen. PUMA: secure inference of llama-7b in five minutes. *CoRR*, abs/2307.12533, 2023.

[8] David Evans, Vladimir Kolesnikov, and Mike Rosulek. *A Pragmatic Introduction to Secure Multi-Party Computation*. 2018.

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

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

[11] Xiaoyang Hou, Jian Liu, Jingyu Li, Yuhan Li, Wen jie Lu, Cheng Hong, and Kui Ren. Ciphergpt: Secure two-party gpt inference. Cryptology ePrint Archive, Paper 2023/1147, 2023. https://eprint.iacr.org/2023/1147.

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

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

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

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

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

[17] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 19–38. IEEE Computer Society, 2017.

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

[19] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: improved mixed-protocol secure two-party computation. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2165–2182. USENIX Association, 2021.

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

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

[22] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. *Des. Codes Cryptogr.*, 71(1):57–81, 2014.

[23] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1607–1626. ACM, 2020.

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

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

A The Definitions of some Activation Functions

- LeakyReLU :

$$\text{LeakyReLU}(x) = \begin{cases} x, & x \geq 0 \\ ax, & x < 0 \end{cases}$$

  where a is a small, positive constant.

  This coefficient ensures that the network can propagate gradients even for negative input values, thereby mitigating the "dying ReLU" problem.

- PReLU :

$$\text{PReLU}(x) = \begin{cases} x, & x \geq 0 \\ \alpha x, & x < 0 \end{cases}$$

  where α is a learnable parameter. PReLU adapts α during training, allowing the network to optimize its activation function based on the data characteristics.

- Hardtanh :

$$\text{Hardtanh}(x) = \begin{cases} 1, & x > 1 \\ -1, & x < -1 \\ x, & \text{otherwise} \end{cases}.$$

- Hardsigmoid :

$$\text{Hardsigmoid}(x) = \max\{0, \min\{1, \alpha x + \beta\}\},$$

  where α and β are constants.

- ReLU6 :

$$\text{ReLU6}(x) = \begin{cases} 6, & x < 6 \\ x, & 0 \leq x < 6 \\ 0 & x < 0 \end{cases}.$$

- Softshrink :

$$\text{Softshrink}(x) = \begin{cases} x - a, & x > a \\ x + a, & x < -a \\ 0, & \text{otherwise} \end{cases}$$

  where a is a constant.

- Softsign :

$$\text{Softsign}(x) = \frac{x}{1 + |x|}.$$

- Hardtanh :

$$\text{Hardtanh}(x) = \begin{cases} 1, & x > 1 \\ -1, & x < -1 \\ x & \text{otherwise} \end{cases}.$$

- Hardshrink :

$$\text{Hardshrink}(x) = \begin{cases} x, & x > a \\ x, & x < -a \\ 0 & \text{otherwise} \end{cases}$$

  where a is a constant.

- Tanhshrink :

$$\text{Tanhshrink}(x) = x - \text{Tanh}(x).$$

B The proof of Theorem 1

Proof 1 *The computation of* $\text{DReLU}(x)$ *has already been proven in CrypTFlow2 [21], so we focus on proving the equality function part. At first,* $\text{int}(x) = -1$ *if and only* $x_0 + x_1 = 2^l - 1$. *As* $x_i = m_i \cdot 2^{l-1} + y_i$ *for* $i \in \{0,1\}$, *we have* $x_0 + x_1 = (m_0 + m_1) \cdot 2^{l-1} + (y_0 + y_1)$. *Therefore,* $x_0 + x_1 = 2^l - 1$ *if and only if* $m_0 + m_1 = 1$ *and* $y_0 + y_1 = 2^{l-1} - 1$, *as* $0 \leq y_0, y_1 \leq 2^{l-1} - 1$. *Finally, we have* $\mathbf{1}\{\text{int}(x) = -1\} = \text{eq} \wedge (m_0 \oplus m_1)$.

C Experimental Results

Experimental results for SEAF and the comparisons to prior works are presented in Table 7 and Table 8. Experiments are implemented using IKNP-style OT in Table 7, and VOLE-style OT in Table 8.

Table 7: The communication cost and runtimes of SEAF on activation functions Tanh, Sigmoid, GELU, and ELU, with IKNP-style OT. The Time (L) and Time (W) denote the runtime under LAN and WAN, respectively. The bitwidth and precision of input x is $l = 21$ and $f = 12$. For $(l_a, l_d) \neq (21, 21)$, the precision is $(f_a, f_d) = (l_a - 1, l_d - 1)$. For $(l_a, l_d) = (21, 21)$, we set $f_a = f_d = f = 12$. The runtime (in seconds) is accumulated for 2^{20} runs of the protocols, and the Comm. rows list the communication costs (in bits) for a single instance.

(a) Tanh

(l_a, l_d)	(21, 21)	(7, 13)	(7, 12)	(6, 12)	(6, 11)	(5, 11)	(5, 10)	(4, 10)	(4, 9)
Comm.	16583	10824	10760	10376	10320	9936	9872	9488	9424
Time (L)	27.76	21.64	21.37	21.28	21.17	21.34	20.89	20.86	20.16
Time (W)	205.21	136.82	136.44	132.50	131.08	127.54	127.09	122.67	122.26
maxULP	3	3	4	5	6	9	10	17	18
avgULP	0.76	0.82	0.83	1.05	1.41	1.67	2.59	3.15	5.53

(b) Sigmoid

(l_a, l_d)	(21, 21)	(9, 13)	(8, 12)	(7, 12)	(6, 13)	(6, 11)	(5, 12)	(5, 11)	(4, 12)
Comm.	16734	11776	11296	10912	10592	10464	10144	10080	9760
Time (L)	28.31	22.61	21.86	21.78	21.32	21.26	20.76	20.65	20.02
Time (W)	206.38	149.97	143.37	138.93	135.18	134.19	130.04	129.82	126.59
maxULP	3	3	4	5	6	7	10	11	17
avgULP	1.11	1.07	1.28	1.32	1.42	1.74	1.90	2.14	2.73

(c) GELU

(l_a, l_d)	(21, 21)	(8, 13)	(6, 13)	(6, 12)	(6, 11)	(5, 11)	(4, 11)	(4, 10)	(4, 9)
Comm.	16881	11264	10496	10432	10368	9984	9600	9536	9472
Time (L)	28.40	22.17	21.78	21.24	21.17	20.63	20.77	20.28	20.12
Time (W)	209.52	142.62	133.85	132.54	132.33	127.68	124.38	123.05	125.10
maxULP	3	3	4	5	6	7	9	12	17
avgULP	1.08	1.09	1.14	1.13	1.36	1.53	2.05	2.59	4.19

(d) ELU $s = 7$

(l_a, l_d)	(21, 21)	(8, 13)	(6, 13)	(6, 12)	(5, 12)	(5, 11)	(5, 10)	(4, 11)	(4, 9)
Comm.	18161	12040	11000	10872	10152	10024	9896	9568	9312
Time (L)	29.31	24.47	23.85	23.56	23.53	22.88	22.61	22.57	22.34
Time (W)	220.42	149.34	139.77	135.81	135.02	126.54	121.45	120.34	120.17
maxULP	2	2	3	4	5	7	9	13	17
avgULP	0.46	0.39	0.37	0.40	0.50	0.68	1.28	0.90	2.95

Table 8: The communication cost and runtimes of SEAF on activation functions Tanh, Sigmoid, GELU, and ELU. The only difference between this table and Table 7 is that the experiments in this table are implemented using VOLE-style OT.

(a) Tanh

(l_a, l_d)	(21, 21)	(7, 13)	(7, 12)	(6, 12)	(6, 11)	(5, 11)	(5, 10)	(4, 10)	(4, 9)
Comm.	5512	2736	2680	2688	2488	2360	2432	2104	2176
Time (L)	31.12	23.96	23.38	23.11	24.07	22.99	22.73	22.12	22.03
Time (W)	100.18	50.89	49.76	47.25	47.57	44.93	43.95	40.90	40.65
maxULP	3	3	4	5	6	9	10	17	18
avgULP	0.76	0.82	0.83	1.05	1.41	1.67	2.59	3.15	5.53

(b) Sigmoid

(l_a, l_d)	(21, 21)	(9, 13)	(8, 12)	(7, 12)	(6, 13)	(6, 11)	(5, 12)	(5, 11)	(4, 12)
Comm.	5568	3096	2848	2856	2648	2664	2456	2392	2408
Time (L)	33.18	26.06	25.30	25.20	24.94	24.29	24.66	24.64	23.39
Time (W)	102.91	56.53	53.34	51.84	49.89	47.59	47.53	46.40	44.16
maxULP	3	3	4	5	6	7	10	11	17
avgULP	1.11	1.07	1.28	1.32	1.42	1.74	1.90	2.14	2.73

(c) GELU

(l_a, l_d)	(21, 21)	(8, 13)	(6, 13)	(6, 12)	(6, 11)	(5, 11)	(4, 11)	(4, 10)	(4, 9)
Comm.	5576	2920	2664	2736	2536	2368	2360	2160	2232
Time (L)	30.78	22.20	22.61	22.23	22.00	21.64	20.03	21.26	20.76
Time (W)	101.66	51.40	48.19	46.82	46.03	44.24	40.72	41.43	39.40
maxULP	3	3	4	5	6	7	9	12	17
avgULP	1.08	1.09	1.14	1.13	1.36	1.53	2.05	2.59	4.19

(d) ELU $s = 7$

(l_a, l_d)	(21, 21)	(8, 13)	(6, 13)	(6, 12)	(5, 12)	(5, 11)	(5, 10)	(4, 11)	(4, 9)
Comm.	8344	4216	3888	3760	3504	3312	3248	3192	2800
Time (L)	39.69	26.73	25.21	24.98	25.05	24.40	24.37	23.54	22.78
Time (W)	159.12	69.08	61.34	59.64	57.38	54.40	53.41	51.67	48.25
maxULP	2	2	3	4	5	7	9	13	17
avgULP	0.46	0.39	0.37	0.40	0.50	0.68	1.28	0.90	2.95