Aegis: A Lightning Fast Privacy-preserving Machine Learning Platform against Malicious Adversaries

Tianpei Lu*, Bingsheng Zhang*, Lichun Li† and Kui Ren*

*The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, Email: {luitianpei, bingsheng, kui.ren}@zju.edu.cn.
†Ant Group Co., Ltd, Hangzhou, China, Email: lichun.lc@antgroup.com

Abstract—Privacy-preserving machine learning (PPML) techniques have gained significant popularity in the past years. Those protocols have been widely adopted in many real-world security-sensitive machine learning scenarios, e.g., medical care and finance. In this work, we introduce Aegis – a high-performance PPML platform built on top of a maliciously secure 3-PC framework over ring \( \mathbb{Z}_{2^\ell} \). In particular, we propose a novel 2-round secure comparison (a.k.a., sign bit extraction) protocol in the preprocessing model. The communication of its semi-honest version is only 25% of the state-of-the-art (SOTA) constant-round semi-honest comparison protocol by Zhou et al. (S&P 2023); both communication and round complexity of its malicious version are approximately 50% of the SOTA (BLAZE) by Patra and Suresh (NDSS 2020), for \( \ell = 64 \). Moreover, the communication of our maliciously secure inner product protocol is merely \( 3\ell/4 \) bits, reducing 50% from the SOTA (Swift) by Koti et al. (USENIX 2021). Finally, the resulting ReLU and MaxPool PPML protocols outperform the SOTA by \( 4 \times \) in the semi-honest setting and \( 10 \times \) in the malicious setting, respectively.

1. Introduction

In the era of big data, privacy protection and compliance continues to be a matter of paramount concern among individuals and organizations alike. With the rise of various privacy regulations, such as GDPR, the need for privacy-preserving mechanisms has intensified. Privacy-preserving machine learning (PPML) is an emerging privacy-enhancing technique that enables secure data mining and machine learning while maintaining the privacy and confidentiality of the underlying data.

Secure multi-party computation (MPC) [1], [17], [38] allows \( n \) parties to jointly evaluate certain functions without revealing their private inputs, and it is a typical cryptographic tool to realize PPML [6], [26], [27], [30], [33], [34], [35] in the multi-server setting. Most of these protocols [8], [34] are designed for the semi-honest setting; whereas, the state-of-the-art (SOTA) maliciously secure PPML protocols suffer a significant performance overhead. For instance, the maliciously secure multiplication protocol of [13], [24] is roughly \( 2 \times \) slower than its semi-honest version.

PPML-friendly MPC protocols usually operate over rings \( \mathbb{Z}_{2^\ell} \) to facilitate the fixed point arithmetics. However, it is more difficult to design maliciously secure MPC over \( \mathbb{Z}_{2^\ell} \) than MPC over a prime-order finite field \( \mathbb{Z}_p \). There have been a series of works such as [16], [19], [28] implementing efficient maliciously secure protocols over \( \mathbb{Z}_p \). Some techniques used in MPC over \( \mathbb{Z}_p \) to achieve malicious security cannot be directly adopted to the MPC over \( \mathbb{Z}_{2^\ell} \) as elements in \( \mathbb{Z}_{2^\ell} \) may not have an inverse. Some attempts [12], [15], [21] have been made, but the resulting protocols come with a \( 2 \times \) communication overhead. Alternatively, another line of works, such as [13], [24], [30] tries to design maliciously secure MPC over \( \mathbb{Z}_{2^\ell} \) from scratch, but their solutions are still significantly slower than the corresponding semi-honest protocols.

Another challenge of PPML is that machine learning algorithms often utilize many non-arithmetic functions, which cannot be efficiently evaluated by MPC. For instance, the activation functions used in machine learning, such as Rectified Linear Unit (ReLU), and MaxPool, extensively use secure comparisons. One possible solution [9], [22], [27], [31] is to mix arithmetic circuits and boolean circuits, evaluating multiplication and addition on the arithmetic circuits and the non-arithmetic functions, e.g., comparison and shift, on the boolean circuits. However, this approach needs costly share conversion between arithmetic and boolean fields. Recently, many SOTA PPML protocols, such as [25], [32], [34], [35], [40], introduce tailor-made protocols to evaluate certain non-arithmetic functions, such as comparison and ReLU, eliminating the need for share conversion.

Our results. In this work, we propose Aegis – a maliciously secure PPML platform that is based on 3-party computation (3PC) in an honest majority setting. The underlying share of our 3PC protocol originates from a variant of the replicated secure sharing [8], and we add 2T-secure MAC to enable fast verification of non-arithmetic functions, such as ReLU. (cf. Tbl. 2, below). In addition, we follow the batch verification paradigm proposed by [19], [28] to verify the correctness of all the multiplication gates at the same time. In particular, we extend the shared elements over \( \mathbb{Z}_{2^\ell} \) [3], [4], [5] to the quotient ring of polynomials \( \mathbb{Z}_{2^\ell}[x]/f(x) \), where \( f(x) \) is a degree-\( d \) irreducible polynomial over \( \mathbb{Z}_2 \) in order to apply the Lagrange interpolating based dimension...
Secure comparison

To achieve

Aegis protocols in and the MaxPool protocol by integrating the above basic
inner product verification protocol Π

maliciously secure positive assertion protocol

the correctness of multiplication
functions, e.g. ReLU and MaxPool. (cf. Table 5 in the
appendix for more details of the communication cost of our
protocols.)

Two-round sign bit extraction. Secure comparison
(a.k.a. sign bit extraction) is essential for PPML. We design
a 2-round comparison protocol that can be further used
to construct the ReLU and MaxPool protocols. Compared
with CryptTFlow [25] (8-round with 6ℓlog ℓ + 14ℓ bits
communication) and Bicoptor [40] (2-round with the
(ℓ + ℓ)(2 + ℓ) bits communication, with error probability
2−1−ε), our protocol demonstrates significant improvements
(2-round with 4ℓ log ℓ + 6ℓ communication). Specifically,
our protocol reduces the communication cost by 75%
for the semi-honest setting. Furthermore, in real-world
benchmark tests, our protocol exhibits 4× speedup over
SOTA.

Sign bit verification with IT-secure MAC. To achieve
maliciously secure sign bit extraction, we adopt SPDZ style
IT-secure MAC [14] and dual execution technique [23].
The resulting protocol only requires a 2-round with 10λℓ(log ℓ +
1) + 14ℓ log ℓ + 16ℓ bits communication while λ is the
statistical security parameter and the soundness error is
2−(λ log ℓ+λ+ log ℓ). To the best of our knowledge, our ma-
liciously secure protocol significantly reduces communica-
tion of SOTA constant round solutions. Compared with
BLAZE [30] (5-rounds with 5κℓ+6ℓ+κ bits communication
in the offline phase and 4-round and κℓ + 6ℓ bits com-
munication in the online phase), our protocol reduces both
the communication and round complexity by 50%, when
ℓ = 64, κ = 128 and λ = 6 (with statistical soundness
error 2−48). In addition, our protocol requires much less
computation than BLAZE which is based on Garble Circuit.

Batch verification for multiplication over ring.

Compared with the prime-order finite field, constructing an
MPC over ring Zp against malicious adversaries typically
incurs a higher overhead. In this work, we propose a
new maliciously secure 3PC multiplication protocol over
ring Zp with a logarithmic communication overhead
during batch verification. We conduct benchmarks on the
overhead ratio of the verification step. As illustrated in
Fig. 15 below, the verification cost is insignificant w.r.t.
the total execution time. By employing this technique, the
amortized communication cost of our maliciously secure
multiplication is merely 2 ring elements in the online phase
and 1 ring element in the offline phase per operation.

Compared with SOTA maliciously secure MPC multipli-
cation over ring [13], our protocol reduces the overall
communication by 40%. Note that [13] achieves full secu-
rity in the Q3 active adversary setting (t < n/3), while
our protocol achieves security with abort in the Q2 active
adversary setting (t < n/2), where t is the number of
corrupted parties and n is the total number of participants.
Compared with SOTA 3PC multiplication over ring [24], our
protocol reduces the communication by 33% in the online
phase and 67% in the offline phase, respectively. Similarly,
the communication of our inner product protocols is also

Our batch multiplication verification protocol Π

and inner product verification protocol Π

serve the
purpose of batch-verifying the correctness of multiplication
triples and inner product triples. These protocols are built
on top of the aforementioned Π

and Π

. Ad-
ditionally, we also develop a maliciously secure truncation
protocol Π

with no online communication. Finally, we
built the convolution protocol, the ReLU protocol Π

and the MaxPool protocol by integrating the above basic
protocols.

Performance. Table 1 depicts the comparison between our
protocols in Aegis and SOTA 3PC-based PPML solutions.

As we can see, Aegis achieves a significant performance
improvement for both multiplication and non-arithmetic
functions.
TABLE 1: Comparison of 3-PC based PPML. ($\ell$ is the ring size, $\ell^*$ is the security parameter for truncation error $2^{1-\epsilon^*}$, $n$ is the size of the inner product, $\kappa = 128$ is the security parameter of GC, and $\lambda = 5$ is the statistical security parameter.)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Protocol</th>
<th>Offline</th>
<th>Online</th>
<th>Malicious</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Protocol</td>
<td>Communication (bits)</td>
<td>Rounds</td>
<td>Communication (bits)</td>
</tr>
<tr>
<td>Mult</td>
<td>ABY3 [27]</td>
<td>12$\ell$</td>
<td>1</td>
<td>9$\ell$</td>
</tr>
<tr>
<td></td>
<td>BLAZE [30]</td>
<td>3$\ell$</td>
<td>1</td>
<td>3$\ell$</td>
</tr>
<tr>
<td></td>
<td>SWIFT [24]</td>
<td>3$\ell$</td>
<td>1</td>
<td>3$\ell$</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>$\ell$</td>
<td>1</td>
<td>2$\ell$</td>
</tr>
<tr>
<td>Inner Product</td>
<td>ABY3 [27]</td>
<td>12$n\ell$</td>
<td>1</td>
<td>9$n\ell$</td>
</tr>
<tr>
<td></td>
<td>BLAZE [30]</td>
<td>3$n\ell$</td>
<td>1</td>
<td>3$n\ell$</td>
</tr>
<tr>
<td></td>
<td>SWIFT [24]</td>
<td>3$n\ell$</td>
<td>1</td>
<td>3$n\ell$</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>$n\ell$</td>
<td>1</td>
<td>2$n\ell$</td>
</tr>
<tr>
<td>Inner Product with Truncation</td>
<td>ABY3 [27]</td>
<td>12$n\ell + 8\ell$</td>
<td>1</td>
<td>9$n\ell + 3\ell$</td>
</tr>
<tr>
<td></td>
<td>BLAZE [30]</td>
<td>3$n\ell + 2\ell$</td>
<td>1</td>
<td>3$n\ell + 3\ell$</td>
</tr>
<tr>
<td></td>
<td>SWIFT [24]</td>
<td>5$n\ell$</td>
<td>1</td>
<td>3$n\ell$</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>$n\ell$</td>
<td>1</td>
<td>2$n\ell$</td>
</tr>
<tr>
<td>ReLU</td>
<td>ABY3 [27]</td>
<td>60$\ell$</td>
<td>4</td>
<td>45$\ell$</td>
</tr>
<tr>
<td></td>
<td>BLAZE [30]</td>
<td>5$\ell + 6\ell + \kappa$</td>
<td>4</td>
<td>$6\ell + 6\ell$</td>
</tr>
<tr>
<td></td>
<td>SWIFT [24]</td>
<td>2$\ell$</td>
<td>2</td>
<td>2$\ell$</td>
</tr>
<tr>
<td></td>
<td>Falcon [35]</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Bicorbit [40]</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Ours (Semi-honest)</td>
<td>$\ell \log_2 \ell + 4\ell$</td>
<td>2</td>
<td>4$\ell \log_2 \ell + 8\ell$</td>
</tr>
<tr>
<td></td>
<td>Ours (Malicious)</td>
<td>$\ell \log_2 \ell + 4\ell$</td>
<td>2</td>
<td>10$\ell \log_2 \ell + 10\ell$</td>
</tr>
</tbody>
</table>

50% of that in SWIFT [24].

2. Preliminaries

Notation. Let $P := \{P_0, P_1, P_2\}$ be the three MPC parties. During the PMML execution, we encode the float numbers as fixed-point structure [27], [30]: for a fixed point value $x$ with $k$-bit precision, if $x \geq 0$, we encode it as $x \cdot 2^k$; if $x < 0$, we encode it as $2^k + x \cdot 2^k$. This encoding method utilizes the most significant bit as the sign bit. For a ring element $x$, the $i^{th}$ bit from big endian is denoted by $x_i$. We denote $y(x) = \alpha \cdot x$ as the MAC of $x$ where $\alpha$ is the MAC key. We denote sign($x$) as the sign bit of $x$ and rshift($x$) as the arithmetic right shift of $x$. Our protocol contains four types of secret sharing as shown in Table 2:

- $[\cdot]$-sharing: We define $[\cdot]$-sharing over ring $Z_{2^t}$ as $\langle x \rangle := (x_1 \in Z_{2^t}, x_2 \in Z_{2^t})$ where $x = x_1 + x_2$. $P_i$ for $i \in \{1, 2\}$ hold share $[x_i]$, and $P_0$ holds the plaintext $x$.

- $\langle \cdot \rangle$-sharing: We define $\langle \cdot \rangle$-sharing over ring $Z_{2^t}$ as $\langle x \rangle := (\{x\} \in Z_{2^t}, m_x \in Z_{2^t})$ where $m_x$ is a fresh random value and $x = m_x + x_1 + x_2$. $P_i$ for $i \in \{1, 2\}$ hold $\{x\}$, $m_x$, and $P_0$ holds $\{x\}$.

- $\{\cdot\}$-sharing: We define $\{\cdot\}$-sharing over finite field $Z_{2^t}$ as $\{x\} := (\{x\} \in Z_{2^t}, m_x \in Z_{2^t})$ where $x = m_x + x_1 + x_2$ (mod $p$). $P_i$ for $i \in \{1, 2\}$ hold share $\{x\}$, and $P_0$ holds the plaintext $x$.

- $\langle \cdot \rangle^{p,\lambda}$-sharing: We define $\langle \cdot \rangle^{p,\lambda}$-sharing over finite field $Z_{2^t}$ as $\langle x \rangle^{p,\lambda} := (\{x\}^{p,\lambda}, \{\alpha_{i}\}^{p,\lambda}, \{\gamma(x)\}^{p,\lambda})_{j \in \lambda}$. In our sign-bit verification protocol, one party $P_i$ holds the plaintext of $(x, \{\alpha_{i}, \gamma(x)\}_{j \in \lambda})$, and the other parties $P_k$ for $k \in \{i + 1 \mod 3, i + 2 \mod 3\}$ hold the share $(\{x\}_{k}, \{\alpha_{i+k}, \gamma(x)_{k}\}_{j \in \lambda})$.

We use $[\cdot]^{\ell \langle x \rangle}$ and $\langle \cdot \rangle^{\ell \langle x \rangle}$ to denote the share in the polynomial ring $Z_{2^t}[x]/f(x)$ where $f(x)$ is a degree-$d$ irreducible polynomial over $Z_{2^t}$. For $\| \cdot \|_{p,\lambda}$ we utilize subscript $i$ to denote that the plaintext is held by $P_i$. Note that we let any two shared values $\|x\|_{p,\lambda}$ and $\|y\|_{p,\lambda}$ for plaintext holder $P_i$ use the same MAC key. For simplicity, we use $\| \cdot \|_{p,\lambda}$ when semantics are clear.

All the aforementioned secret-sharing forms have the linear homomorphic property, i.e., $\|x + y\| = \|x\| + \|y\|$. $\|x\|^{\ell \langle x \rangle}$ and $\langle x \rangle^{\ell \langle x \rangle}$ for $\| \cdot \|$, we have $\|x\| + \|y\| = \|x\| + \|y\|$. $\|\gamma(x)\| \gamma(x)_j \in \lambda$ and $\|\cdot\| = \|\cdot\| + \|\gamma(x)\| \gamma(x)_j \in \lambda$.

Secret sharing. Let $\Pi_1(x)$, $\Pi_1(x)$, and $\Pi_1(x)$ denote the corresponding secret sharing protocols. By $\Pi_1(x)$, we mean that $x$ is shared by $P_0$; by $\Pi_1(x)$, we mean the parties jointly generate a shared random value. We utilize pseudo-random generators (PRG) to reduce the communication [39]. In our protocol description, when we let several parties pick the same random values together, we mean that these parties use PRG to locally generate random values with an agreed-upon seed. The brief sketch of secret sharing schemes are as follows.

- $x \leftarrow \Pi_1(x)$:
  - $P_0$ and $P_1$ pick random value $x_1 \in Z_{2^{t'}}, x_2 \in Z_{2^{t'}}$;
  - $P_0$ sends $x_2 = x - x_1$ (mod $2^{t'}$) to $P_2$.

- $x \leftarrow \Pi_1(x)$:
  - $P_0$ and $P_1$ pick random value $x_1 \in Z_{2^{t'}}$;
  - $P_0$ and $P_2$ pick random value $x_2 \in Z_{2^{t'}}$;
  - $P_0$ calculates $x = x_1 + x_2$.

- $\|x\| \leftarrow \Pi_1(x)$:
TABLE 2: The share structure of Aegis. (For $\| \cdot \|_p$, the example in the table depicts the case of $\| \cdot \|_{0,p}$)

<table>
<thead>
<tr>
<th>$P$</th>
<th>$[x]^p$</th>
<th>$x$ (or $\langle x \rangle_{Z_p}$)</th>
<th>$[x]_0^p$</th>
<th>$x$ (or $\langle x \rangle_{Z_p}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>$x$</td>
<td>$\langle x, \alpha \rangle_{Z_p}, \langle \gamma(x) \rangle_{Z_p}$</td>
<td>$x$</td>
<td>$\langle x \rangle_{Z_p}$</td>
</tr>
<tr>
<td>$P_1$</td>
<td>$\langle [x]_0^p \rangle \in Z_p$</td>
<td>$\langle [x]<em>0^p, \langle \alpha \rangle</em>{Z_p}, \langle \gamma(x) \rangle_{Z_p} \rangle \in Z_p$</td>
<td>$\langle [x]_0^p \rangle \in Z_p$</td>
<td>$\langle [x]_0^p \rangle \in Z_p$</td>
</tr>
<tr>
<td>$P_2$</td>
<td>$\langle [x]_0^p \rangle \in Z_p$</td>
<td>$\langle [x]<em>0^p, \langle \alpha \rangle</em>{Z_p}, \langle \gamma(x) \rangle_{Z_p} \rangle \in Z_p$</td>
<td>$\langle [x]_0^p \rangle \in Z_p$</td>
<td>$\langle [x]_0^p \rangle \in Z_p$</td>
</tr>
</tbody>
</table>

For the share $\langle \cdot \rangle^i$ in polynomial ring, $\Pi_{\text{Rec}}^i$ works analogously as the above.

**Preprocessing and postprocessing.** We follow the “preprocessing” paradigm [2] which splits the protocol into two phases: the preprocessing/offline phase is data-independent and can be executed without data input, and the online phase is data-dependent and is executed after data input. Specifically, all the items $r_x$ of share $\langle x \rangle$ of our protocols can be generated in the circuit-depend offline phase. What the parties need to do in the online phase is to collaborate in computing $m_x$ for $P_1$ and $P_2$. To achieve malicious security, we further introduce the postprocessing phase [21] where batch verification is performed.

**Multiplication gate.** We adopt the multiplication protocol of ASTRA [8]. For multiplication $z = x \cdot y$ with input $\langle x \rangle$, $\langle y \rangle$ and output $\langle z \rangle$, all parties first generate $\langle r_x \rangle \leftarrow \Pi(\langle r \rangle)$ for the output wire in the offline phase. To calculate $m_x$ for $P_1$ and $P_2$ in the online phase, it can be written as

$$m_x = [x y] = \sum_{j=1}^{n} [x_j y_j] = [x y]_0 \cdot [x y]_1 + \ldots + [x y]_n$$

where $[x y]_j$ is the $j$-th item of $[x]_{[y]}$. For $P_1$ and $P_2$ locally and $[\Gamma] = [x y] - [r_x]$ can be secret shared by $P_0$ to $P_1$ and $P_2$ in the preprocessing phase. In the online phase, $P_1$ and $P_2$ calculate and reconstruct $[m_x] = [\Gamma] + [\Gamma']$.

**Multivariate polynomial evaluation.** Given a $d$-degree $n$-variate polynomial function $F^d(x_1, \ldots, x_n) = y$, we design a evaluation protocol $\langle y \rangle = \Pi_{\text{PolyEvl}}(F^d(x_1, \ldots, x_n))$ which requires communication of $2d$ bits in the online phase and at most $d \cdot (n^d - 1)$ bits in the offline phase. In particular, plug the underlying shares, we have

$$m_y = F^d(m_{x_1}, r_{x_1}, \ldots, m_{x_n} - r_{x_n}) + r_y$$

where $m_y$ is the output of the polynomial evaluation. After expanding Eq. 1, we let $P_0$ locally computes all the cross-terms $\Pi_{x_s \in Z_p, x_s \in Z_p} r_{x_s}$ and share them to the other parties in the offline phase. The offline phase requires $\ell \cdot m$ bits communication depending on the number of cross-terms, i.e. $m$. Let $\Pi_{\text{PolyEvl}}^i$ denote the polynomial evaluation protocol w.r.t. a polynomial ring $\Z[x] / f(x)$. Analogously, it costs $2d \cdot m$ in the communicating phase, for the degree $d$ of $f(x)$.

**Security up to additive attacks.** As proven in [10], an replicated secret sharing protocol, such as $\Pi_{\text{PolyEvl}}$, is secure up to additive attacks against malicious adversaries, i.e., the adversary’s cheating ability is limited to introducing an additive error to the output.
3. Security Model

We analyze the security of our protocols in the well-known Universal Composability (UC) framework [7], which follows the simulation-based security paradigm. The adversary \( A \) is allowed to partially control the communication tapes of all uncorrupted machines, that is, it sees all the messages sent from and to the uncorrupted machines and controls the sequence in which they are delivered. Then, a protocol \( \Pi \) is a secure realization of the functionality \( F \), if it satisfies that for every PPT adversary \( A \) attacking an execution of \( \Pi \), there is another PPT adversary \( S \) (simulator) attacking the ideal process that uses \( F \) where the executions of \( \Pi \) with \( A \) and that of \( F \) with \( S \) makes no difference to any PPT environment \( Z \).

The idea world execution. In the ideal world, the parties \( \mathcal{P} := \{ P_0, P_1, P_2 \} \) only communicate with the ideal functionality \( F_{\text{3pc}} \) with the excited function \( f \). All parties send their share to \( F_{\text{3pc}}, F_{\text{3pc}} \) calculate and output the result depend on the adversary \( S \).

Functionality \( F_{\text{3pc}} \)

\( F_{\text{3pc}} \) interacts with the parties in \( \mathcal{P} \) and the adversary \( S \). Let \( f \) denote the functionality to be computed.

**Input:**
- Upon receiving from (Input, sid, \( x_i \)) from \( P_i \) in \( \mathcal{P} \), record \( x_i \) and send (Input, sid, \( P_i \)) to \( S \).
- For \( i \in [3] \), send (Output, \( y_i \)) to \( P_i \) via private delayed channel.

**Execution:**
- Upon receiving (Compute, sid) from \( S \), if all \( x_i \) are recorded compute \((y_0, y_1, y_2) = f(x_0, x_1, x_2)\).

Figure 2: The ideal functionality \( F_{\text{3pc}} \).

The real world execution. In the real world, the parties \( \mathcal{P} := \{ P_0, P_1, P_2 \} \) communicate with each other via secure channel functionality \( F_{\text{3pc}} \) for the protocol execution \( \Pi \). Our protocols work in the pre-processing model, but, for simplicity, we analyze the offline and online protocols together as a whole.

**Definition 1.** We say protocol \( \Pi \) UC-secure realizes functionality \( F \) if for all PPT adversaries \( A \) there exists a PPT simulator \( S \) such that for all PPT environment \( Z \) it holds:

\[
\text{Real}_{\Pi,A,Z}(1^\lambda) \approx \text{Ideal}_{F,S,Z}(1^\lambda)
\]

4. Secure Sign Bit Extraction

In this section, we propose a new sign bit extraction protocol \( \Pi_{\text{SignBit}} \). For sign bit extraction function \( y = \text{sign}(x) \), protocol \( \Pi_{\text{SignBit}} \) can output \((y)\) from input \((x)\). In Sec. 5, we apply it to the malicious setting.

Let \( L_1 := \{ s_{ij} \}_{j \in \mathbb{Z}_p} \) be the list of the individual bits of the shared value \( s \). One can transfer \( L_1 \) into another list \( L_2 := \{ t_{ij} \}_{j \in \mathbb{Z}_p} \) such that only one zero-element exists at the position corresponding to the first non-zero bit of \( L_1 \). Namely, \( t_{ij} = \sum_{k=0}^{\ell} s_{ik} 2 \cdot s_{ij} + 1 \mod p \) for \( j \in \mathbb{Z}_p \), where \( p \geq \ell \). We denote this transform as \( \phi \). The intuition of our sign bit extraction is as follows.

Let \( m_x := \text{sign}(m_x) \cdot m_x \) and \( r_x := \text{sign}(-r_x) \cdot r_x \).

Instead of extracting the sign bit, we evaluate \( \text{sign}(x) := (m_x + r_x)^2 - 2^\ell \geq 2^\ell - 1 \geq r_x \).

To calculate \( m_x + r_x \), we evaluate \( m_x \geq 2^\ell - 1 - r_x \). (It works due to \( 2^{\ell - 1} \geq 1 \geq r_x \). If we consider \( m_x \) and \( 2^{\ell - 1} - 1 - r_x \) as a pair of XOR shares of \( m \), the first non-zero bit (denoted its index as ind) of the \( m \) corresponds to the first differing bit between \( m_x \) and \( 2^{\ell - 1} - 1 - r_x \), which indicates the position where the two values diverge. We have \( m_x|_{\text{ind}} = m_x \geq 2^\ell - 1 - r_x \).

Following this intuition, we apply \( \{ m_{ij} \}_{j \in \mathbb{Z}_p} \) to identify the first nonzero bit. As \( \phi \) operates over field \( \mathbb{Z}_p \), we first transfer \( \{ m_{ij} \} \to \mathbb{Z}_p \) to shares over
Figure 4: Compression of Multiplication Triples.

\[ \text{Protocol } \Pi_{\text{trans}} (\{(x_i), (y_i), (z_i)\}_{i \in \mathbb{Z}_N}) \]

Input: \( N \) tuples of \((\cdot)\)-shared multiplication.
Output: One triple of \( N \)-dimension \((\cdot)\)-shared inner product.

Preprocessing:
- All parties invoke \((r)^j \leftarrow \Pi_{\text{trans}}^j\) locally;

Online:
- All parties reconstruct \( r \) with \( \Pi_{\text{rec}} \) and calculate \( r^j \) for all \( i \in \mathbb{Z}_N \);
- All parties transfer \((\cdot)\) to \((\cdot)^j\) locally by setting the constant term of \((\cdot)^j\) to \((\cdot)\);
- All parties set \( \langle x_j \rangle^j = \sum_{j=0}^{N-1} r^j \cdot \langle x_j \rangle^j \) and \( \langle x_j \rangle^j = r^j \cdot \langle x_j \rangle^j \) for all \( i \in \mathbb{Z}_N \);
- All parties output \( \{\langle x_j^j \rangle^j, \langle y_j \rangle^j \}_{i \in \mathbb{Z}_N}; \langle z_j \rangle^j \) .

\[ \text{Protocol } \Pi_{\text{Reduce}} (\{(x_i)^j, (y_i)^j \}_{i \in \mathbb{Z}_N}; (z)^j) \]

Input: \( N \)-dimension \((\cdot)^j\)-shared inner product.
Output: \( N/2 \)-dimension \((\cdot)^j\)-shared inner product.

Execution:
- For \( i \) \( \in \mathbb{Z}_N/2 \), all parties set
  - \( \langle f(0) \rangle^j = \langle x_2 \rangle^j \cdot \langle f(1) \rangle^j = \langle x_2 \rangle^j + 1 \); \( \langle f(2) \rangle^j = 2 \cdot \langle f(1) \rangle^j - \langle f(0) \rangle^j \);
  - \( \langle g(0) \rangle^j = \langle x_2 \rangle^j \cdot \langle g(1) \rangle^j = \langle x_2 \rangle^j + 1 \);
  - \( \langle g(2) \rangle^j = 2 \cdot \langle g(1) \rangle^j - \langle g(0) \rangle^j \);
  - \( \langle h(0) \rangle^j = \langle f(0) \rangle^j \cdot \langle g(0) \rangle^j \);
  - \( \langle h(2) \rangle^j = \langle f(2) \rangle^j \cdot \langle g(2) \rangle^j \);
- All parties invoke \((\cdot)^j \leftarrow \Pi_{\text{Reduce}}^j\) and reveal \( (2 \cdot \langle \cdot \rangle)^j \);
- All parties calculate
  - \( \langle h(\cdot) \rangle^j = \sum_{i=0}^{(\Pi_{\text{Reduce}}^j)^j \cdot \langle \cdot \rangle^j} \langle h(\cdot) \rangle^j \);
  - \( \langle f(\cdot) \rangle^j = \sum_{i=0}^{(\Pi_{\text{Reduce}}^j)^j \cdot \langle \cdot \rangle^j} \langle f(\cdot) \rangle^j \);
  - \( \langle g(\cdot) \rangle^j = \sum_{i=0}^{(\Pi_{\text{Reduce}}^j)^j \cdot \langle \cdot \rangle^j} \langle g(\cdot) \rangle^j \);
- All parties output \( \{\langle f(\cdot) \rangle^j, \langle g(\cdot) \rangle^j \}_{i \in \mathbb{Z}_N}; (h(\cdot))^j \).

\[ \Delta = 0 \]

Our sign bit Extract protocol \( \Pi_{\text{SignBit}} \) costs \( 1 \) round with communication of \( \log \ell + 3f \) bits in the offline phase and requires 2 rounds with communication of \( 4f \log \ell + 6f \) bits in the online phase.

Security. We analyze the security of our sign-bit extraction protocol in the UC framework. The functionality \( \mathcal{F}_{\text{SignBit}} \) for sign bit extraction is defined as follows. As an instantiation of \( \mathcal{F}_{\text{sig}} \) depicted in Fig. 2, \( \mathcal{F}_{\text{SignBit}} \) receives \((\text{Input}, \text{id}, \text{r}_{\text{z}})\) from \( P_0 \), \((\text{Input}, \text{id}, \text{m}_{\text{z}})\) from \( P_1 \), and \((\text{Input}, \text{id}, \text{m}_{\text{z}})\) from \( P_2 \). It calculates \( z = \text{sign}(\text{m}_{\text{z}} - \text{r}_{\text{z}}) \). If \( P_0 \) is corrupted, \( \mathcal{F}_{\text{SignBit}} \) obtains \( [r_{\text{z}}]_1 \) and \( [r_{\text{z}}]_2 \) from \( S \). If \( P_2 \) is corrupted, \( \mathcal{F}_{\text{SignBit}} \) obtains \( [r_{\text{z}}]_1 \) from \( S \) and picks random value \( [r_{\text{z}}]_2 \in \mathbb{Z}_{2f} \); \( \mathcal{F}_{\text{SignBit}} \) sets \( m_{\text{z}} = z + [r_{\text{z}}]_1 + [r_{\text{z}}]_2 \) and sends \((\text{Output}, [r_{\text{z}}]_1, [r_{\text{z}}]_2)\) to \( P_0 \), \((\text{Output}, [r_{\text{z}}]_1, m_{\text{z}})\) to \( P_1 \), and \((\text{Output}, [r_{\text{z}}]_2, m_{\text{z}})\) to \( P_2 \).

Theorem 1. Let PRF, PRF, PRF be the secure pseudo-random functions. The protocol \( \Pi_{\text{SignBit}} \) as depicted in Fig. 3 UC realizes \( \mathcal{F}_{\text{SignBit}} \) against semi-honest PPT adversaries who can statically corrupt up to one party.
Protocol $\Pi_{\text{InnerVerify}}\left(\{(x_i), (y_i), (z_i)\}_{i \in \mathbb{Z}_N}\right)$

**Input:** A $N$-dimension $(\cdot)^{f(x)}$ shared inner product pair.
**Output:** $z = \frac{1}{r} \sum_{i=1}^{N} x_i \cdot y_i$.
**Execution:**
- All parties invoke $(\alpha)^{f(x)} \leftarrow \Pi_{\langle \cdot \rangle}^{f(x)}$.
- All parties calculate $(\Delta)^{f(x)} = (\alpha)^{f(x)} \cdot \left(\sum_{i=1}^{N} (x_i)^{f(x)} \cdot (y_i) - (y_i)^{f(x)}\right)$ with $\Pi_{\text{PolyEval}}$.
- All parties output $\Delta = \Pi_{\text{Rec}}^{f(x)}(\Delta)^{f(x)}$.
- All parties output 1 if $\Delta = 0$, otherwise 0.

Figure 6: The Inner Product Verification Protocol

**Proof.** See Appendix A.1.

5. Achieving Malicious Security

Aegis uses the postprocessing verification procedure to detect any potential malicious behavior. In this section, we first present our batch verification protocol for multiplication and then introduce our verification protocol for the correctness of sign bit extraction.

5.1. Batch Multiplication Verification

We would like to reduce the task of verifying $N$ triples of multiplication $\{(x_i), (y_i), (z_i)\}_{i \in \mathbb{Z}_N}$ to the task of verifying the inner product $\Delta = \sum_{i=0}^{N} (r^i \cdot x_i \cdot (y_i) - (r^i \cdot z_i)$ equals to 0, where $r$ is a fresh random challenge. However, there are two issues with this naive approach. The first issue is that the adversary is aware of the additive error in $(z_i)$, allowing her to cancel out the error when computing $\Delta$ to fabricate $\Delta = 0$. The second issue arises from the irreversible multiplication over the ring, where the adversary can intentionally introduce a specific error $e$ in $z_i$, leading to a high probability of $e \cdot r^i = 0$ to pass the verification. For instance, the adversary can introduce an error $e = 2^{-i}$ in such a way that the equation $r^i \cdot (z_i + e) = r^i \cdot z_i$ holds with a probability of $1/2$ in the case where $r$ is an even number.

To address the former issue, we let all parties evaluate $\Delta = \langle \alpha \rangle \cdot \left(\sum_{i=0}^{N} (r^i \cdot x_i \cdot (y_i) - (r^i \cdot z_i)\right) (using $\Pi_{\text{PolyEval}}$) with random share $(\alpha)$. Since $\Pi_{\text{PolyEval}}$ is secure up to additive attack [10], the adversary can only introduce an input-independent additive error $e'$ of $\Delta$. Therefore, the adversary has to guess $e' = e \cdot \alpha$ and make $\Delta = 0$ with the probability $2^{-d}$. To resolve the latter issue, we perform $\Delta$ over the extension ring $\mathbb{Z}_d[x]/f(x)$, where $f(x)$ is a degree-$d$ irreducible polynomial over $\mathbb{Z}_2$ [3]. (This can be done by putting the original share over $\mathbb{Z}_d[x]$ to be the free coefficient and adding random $d$ elements to the other coefficients.) The probability that a N-degree non-zero polynomial $\Delta(r) = 0$ with a randomly chosen $r$ is at most $2^{\frac{d-1}{2} \cdot \log N}$ by the Schwartz-Zippel Lemma. We further apply the dimension reduction technique of [19] to our ring setting which reduces the $\Theta(N)$ communication of batch verification to $\Omega(\log N \cdot d)$. Following this idea, we construct our batch multiplication verification protocol as follows.

**Compression of multiplication triples.** We first design a subprotocol $\Pi_{\text{Trans}}$ (Fig. 4) which can convert $N$ multiplication triples over $\mathbb{Z}_{2^d}$ to be verified to an $N$-dimension inner product over polynomial ring $\mathbb{Z}_{2^d}[x]/f(x)$. We first transform the multiplication triples $\{(x_i), (y_i), (z_i)\}_{i \in \mathbb{Z}_N}$ to the polynomial ring $\{(x_i)^{f(x)}\}_{i \in \mathbb{Z}_N}$ locally. In this transformation, the free coefficient of the shares over $\mathbb{Z}_d[x]/f(x)$ is set to the original shares and other coefficients are padded with random values. Then, all parties generate a random challenge $r \in \mathbb{Z}_{2^d}[x]/f(x)$ by invoking $(r)^{f(x)} \leftarrow \Pi_{\langle \cdot \rangle}^{f(x)}$ and reconstructing it via $\Pi_{\text{Rec}}$. All parties locally calculate $(z)^{f(x)} = \sum_{i=0}^{N} r^i \cdot (z_i)^{f(x)}$, and $(x')^{f(x)} = r^i \cdot (x_i)^{f(x)}$ for all $i \in \mathbb{Z}_{2^d}$ and return the $N$-dimension inner product tuple as $\{(x)^{f(x)}\}_{i \in \mathbb{Z}_N}, (y)^{f(x)}\}_{i \in \mathbb{Z}_N}, (z)^{f(x)}\}_{i \in \mathbb{Z}_N}$.

**Lemma 1.** Suppose protocol $\Pi_{\text{Trans}}$ depicted in Fig. 4 take input as $\{(x_i), (y_i), (z_i)\}_{i \in \mathbb{Z}_N}$, and it outputs $\{(x')^{f(x)}\}_{i \in \mathbb{Z}_N}, (y)^{f(x)}\}_{i \in \mathbb{Z}_N}, (z)^{f(x)}\}_{i \in \mathbb{Z}_N}$. The probability that the following two conditions hold is at most $\frac{N}{2^d}$, where $d$ is the degree of $f(x)$ w.r.t. $\mathbb{Z}_{2^d}[x]/f(x)$:

- $z = \sum_{i=0}^{N} x'_i \cdot y_i$
- $\exists i \in \mathbb{Z}_{2^d}$ s.t. $z_i \neq x'_i \cdot y_i$

**Proof.** See Appendix A.2.

**Dimension reduction.** We extend the dimension reduction technique of [19] to our 3PC over ring setting. As shown in Fig. 5, protocol $\Pi_{\text{Reduce}}$ takes input as a shared triple $\{(x_i)^{f(x)}\}_{i \in \mathbb{Z}_{2^d}}, (y)^{f(x)}\}_{i \in \mathbb{Z}_N}, (z)^{f(x)}\}_{i \in \mathbb{Z}_N}$ and outputs the shared triple $\{(x''_i)^{f(x)}\}_{i \in \mathbb{Z}_{2^d}}, (y'')^{f(x)}\}_{i \in \mathbb{Z}_N}, (z'')^{f(x)}\}_{i \in \mathbb{Z}_N}$. $\Pi_{\text{Reduce}}$ ensures that $\sum_{i=0}^{N} x''_i \cdot y''_i = z$ if and only if $\sum_{i=0}^{N} x'_i \cdot y'_i = z$ except for a negligible probability. At a high level, for the inner product input $\{x_i\}_{i \in \mathbb{Z}_{2^d}}$ and $\{y_i\}_{i \in \mathbb{Z}_N}$, we can utilize $x_{2i}$ and $x_{2i+1}$ to interpolate $N/2$ linear functions $\{f_i(.)\}_{i \in \mathbb{Z}_{2^d}}$ at the point 0 and 1, and similarly interpolate $\{g_i(.)\}_{i \in \mathbb{Z}_{2^d}}$ by $\{y_i\}_{i \in \mathbb{Z}_N}$. Considering the correct output $z$, we have $z = \sum_{i=0}^{N/2} f_i(0) \cdot g_i(0) + f_i(1) \cdot g_i(1)$. Denote $h(.) = \sum_{i=0}^{N/2} f_i(.) \cdot g_i(.)$. The above equation can be written as...
Theorem 2. Suppose protocol $\Pi_{\text{Reduce}}$ depicted in Fig. 5 take input as $\left\{\left(x_i, e_i[x_i]\right)\right\}_{i \in \mathbb{Z}_N^2}$ and it outputs $\left\{\left(x_i', e_i'[x_i]\right)\right\}_{i \in \mathbb{Z}_N^2}$, where $z$ is the degree of $f(x)$ w.r.t. $\mathbb{Z}_{2^{d'}}[x]/f(x)$. The probability that the following two conditions hold is at most $\frac{1}{2^{d'}}$, where $d$ is the degree of $f(x)$ w.r.t. $\mathbb{Z}_{2^{d'}}[x]/f(x)$:

- $z' = \sum_{i=0}^{N/2} x_i' \cdot y_i'$
- $z \neq \sum_{i=0}^{N/2} x_i \cdot y_i$

Proof. See Appendix A.3.

5.2. Sign Bit Extraction Verification Protocol

We upgrade the sign bit extraction protocol $\Pi_{\text{Reduce}}$ to the malicious setting throughout the verification protocol $\Pi_{\text{SignBit}}$. For a sign bit extraction pair $\{\langle x \rangle, \langle z \rangle\}$, the malicious adversary can introduce arbitrary errors to make $\text{sign}(x) \neq b$. As shown in Fig. 9, we design the verification protocol $\Pi_{\text{SignBit}}$ to verify the correctness of the sign bit extraction pair.

Specifically, the verification consists of two steps: (i) $z$ is validated to be either 0 or 1, (ii) $x - 2^d \cdot z$ is positive. The former check can be realized by employing a maliciously secure multiplication protocol to confirm that
its square matches itself, i.e., \( z \cdot z = z \) on the ring \( \mathbb{Z}_{2^l} \), as \( z^2 - z = 0 \) only has the roots of 0 and 1 over ring \( \mathbb{Z}_{2^l} \). For this check, we directly utilize the aforementioned protocol \( \Pi_{\text{MultiVerify}}((x), (y), (z)) \).

For the latter check, we first design the positive assertion protocol \( \Pi_{\text{Pos}} \) which nominates a verifier \( P_i \) to verify the positive of a shared value. \( \Pi_{\text{Pos}} \) has the property that the honest verifier outputs the correct verification result against one malicious adversary corrupting one of the other two parties. Our protocol is designed for static corruption. To resolve the case where the nominated verifier is malicious, we adopt the dual-execution paradigm \([20],[23]\) to invoke \( \Pi_{\text{Pos}} \) twice with two distinct parties to play the role of the verifier. As the malicious adversary can only statically corrupt one party, we can ensure that the shared value is positive if both two verifications pass.

**Positive assertion protocol** \( \Pi_{\text{Pos}} \). As depicted in Fig. 8, the positive assertion protocol \( \Pi_{\text{Pos}} \) let verifier \( P_i \) (any \( i \in \{0, 1, 2\} \)) take input as shared value \( \langle x \rangle \), and the verifier outputs a bit indicating whether \( x \geq 2^\ell \). Specifically, we introduce the IT-secure MAC to detect malicious behavior of \( P_{1-1} \) and \( P_{1+1} \). We observe that the chopped shared bit \( \| r_{x,j} \| \) in \( \Pi_{\text{SignBit}} \) can be replaced by \( \| r_{x,j} \| \). We let the presumably honest verifier \( P_i \) locally calculate the MAC of \( r_{x,j} \) and secret share it to the other two parties \( P_{1-1} \) and \( P_{1+1} \). Later, when \( P_{1-1} \) and \( P_{1+1} \) send back the opened vector \( \left\{ \| u_j \| \right\}_{j \in \mathbb{Z}_{2^\ell}} \) and \( \left\{ \| u'_j \| \right\}_{j \in \mathbb{Z}_{2^\ell}} \), \( P_i \) can check the correctness of them by the corresponding MAC.

To support dual execution of \( \Pi_{\text{Pos}} \) with different parties playing the role of the verifier, we need to convert the underlying shares accordingly. That is, we express the \( \langle \cdot \rangle \) shared value in the form of replicated secret sharing, which is \( \langle x_0 = m_0, x_1 = -[r_x]_1, x_2 = -[r_x]_2 \rangle \). Following that all parties perform same operation in \( \Pi_{\text{SignBit}} \) which replace \( [r_{x,j}] \) with \( \| r_{x,j} \| \) to generate the the vector \( \left\{ \| u_j \| \right\}_{j \in \mathbb{Z}_{2^\ell}} \) and \( \left\{ \| u'_j \| \right\}_{j \in \mathbb{Z}_{2^\ell}} \). Since \( \text{sign}(x) = 0 \) is public knowledge rather than a secret, we do not need to mask the sign bit at the end. Subsequently, \( P_i \) reconstruct \( u_j, u'_j \) with MAC check, and verify the aforementioned predicate \( \exists u_j = 0 \land u'_j \neq 0 \). The soundness error of \( \Pi_{\text{Pos}}^\lambda \) is \( \frac{1}{2 \times \log^2 \ell \times \log \log_{\lambda} \ell} \).

**Theorem 3.** Let \( \langle x \rangle^\ell \) be the input of the protocol \( \Pi_{\text{Pos}}^\lambda \) depicted Fig. 8. The probability that \( \Pi_{\text{Pos}}^\lambda \) outputs 1 and \( \text{sign}(x) = 1 \) is at most \( \frac{1}{2 \times \log^2 \ell \times \log \log_{\lambda} \ell} \).

**Proof.** See Appendix A.6.

---

### 6. The Aegis PPML Platform

In this section, we present our privacy-preserving machine learning platform Aegis. We start with the construction of our maliciously secure multiplication protocol, such as inner product and convolution. We then utilize the sign bit extraction protocol to construct maliciously secure ReLU and Maxpool protocols.

**Multiplication.** Our maliciously secure multiplication protocol is shown in Fig. 10. \( \Pi_{\text{MultiVerify}} \) ensures the correctness of multiplication by invoking batch verification protocol \( \Pi_{\text{MultiVerify}} \) in the post-processing phase. When handling a substantial volume of data, our protocol exhibits an amortized communication of \( \ell \) bits in the preprocessing phase and \( 2\ell \) bits in the online phase for each multiplication operation.

**Inner product and convolution.** Our maliciously secure inner product protocol \( \Pi_{\text{Inner}} \) is shown in Fig. 12. Its semi-honest version is the special case of \( \Pi_{\text{PolyEval}} \) for 2-degree \( n \)-variate polynomial which requires one round communication of \( \ell \) bits in the preprocessing phase and one round communication of \( 2\ell \) bits in the online phase. To extend it to
the malicious setting, we employ batch verification protocol $\Pi^R_{\text{inner}}$ (Fig. 11) to ensure the correctness of the inner products with a similar manner of multiplication. Analogously, in $\Pi^R_{\text{inner}}$, all parties transform the verification of inner product triples over ring $\mathbb{Z}_{2^\ell}$ to the verification of a single inner product triple over the polynomial ring $\mathbb{Z}_{2^\ell}[x]/f(x)$. Following that, all parties invoke $\Pi_{\text{Reduce}}$ to reduce the dimension of the vector that needs to be verified. When handling a substantial volume of data, on average, our protocol exhibits an amortized communication of $\ell$ bits in the preprocessing phase and $2\ell$ bits in the online phase for each inner product operation. In the application of machine learning, we view the $m$-dimensional output convolution and matrix multiplication as $m$ separate inner products. We implement these two types of operations by invoking $\Pi_{\text{inner}}$ a total of $m$ times.

### Truncation

The multiplication of two fixed-point values with our encoding will lead to a double scale of $2^k$ for the fractional precision $k$. An array of protocols [24, 27, 30] using the probabilistic truncation protocol to reduce the additional $2^k$ scaler. Their protocols introduce a one-bit error which is caused by the carry bit of truncated data. In addition, the probabilistic truncation protocol makes an error with a certain probability (assuming that the valid range of data is $\ell_x$ and the error probability is $2^{\ell_x-\ell+1}$). As shown in Fig. 13, we also design a maliciously secure probabilistic truncation protocol $\Pi^T_{\text{trunc}}$ for the truncation bit size $t$. Our idea is similar to SWIFT [24] which generates correct truncation pair via maliciously secure inner product protocol. However, in contrast to SWIFT [24], we directly generate $r_z = \text{rshift}(r_x, d)$, which allows the parties locally truncate $m_z = \text{rshift}(m_x, d)$ in the online phase without communication. Although SWIFT [24] eliminates communication by combining truncation with multiplication, they still need $2\ell$ online communication in the online phase of the standalone truncation protocol. Specifically, we let $P_0$ and $P_1$ pick random bit list $\{b_{ij}\} \in \mathbb{Z}_2$, together; $P_0$ and $P_2$ pick random bit list $\{b_{ij}\} \in \mathbb{Z}_2$, together. We utilize these lists to compute that

\[
\begin{align*}
    r_x = \sum_{j=0}^{t-1} 2^j \cdot (b_{1j} \oplus b_{2j}) \\
    r_z = \sum_{j=0}^{t-1} 2^j \cdot (b_{1j} \oplus b_{2j} + \sum_{j=t-1}^{t-1} 2^j \cdot (b_{1j-1} \oplus b_{2j-1})
\end{align*}
\]

which keeps the relationship $r_z = \text{shift}(r_x, t)$. We can evaluate $r_x$ and $r_z$ under $(-\cdot)$-sharing to realize malicious security. To transform $b_{1j}$ and $b_{2j}$ to the $(-\cdot)$-sharing locally, we let $b_{1j} = (0, b_{1j})$ and $b_{2j} = (0, b_{2j})$ which set the other secret shard to be 0. For the result ($r_x$) and ($r_z$), since $r_x$ and $r_z$ is known by $P_0$, $P_1$ and $P_2$ can be locally calculate $[r_x] = m_{x_n} - [r_z]$ and $[r_x] = m_{x_n} - [r_z]$. Note that $\Pi_{\text{trunc}}$ requires assigning $r_x$ of the input wire, we let it be executed preferentially to provide $r_x$ for the other gate. Our maliciously secure protocol $\Pi_{\text{trunc}}$ requires 1 rounds and communication of $6\ell t$ bits in the offline phase and requires no communication in the online phase. The semi-honest version of truncation is provided in Appendix B.1

### ReLU

The ReLU of $x$ is calculated by $w = x \cdot (1 - \text{sign}(x)) = x - x \cdot \text{sign}(x)$, which can be implemented by combining $\Pi_{\text{Mult}}$ with $\Pi_{\text{SignBit}}$. However, it requires an additional round for multiplication. We observe that the additional round can be eliminated by executing multiplication at the same round of sending back $m''$ in $\Pi_{\text{SignBit}}$. We construct the semi-honest ReLU protocol $\Pi_{\text{ReLU}}$ (Fig. 14) from $\Pi_{\text{SignBit}}$. Considering $(z) = \Pi_{\text{SignBit}}(\langle x \rangle)$ and $w = \Pi_{\text{Mult}}(\langle x \rangle \cdot \langle z \rangle)$, we have:

\[
\begin{align*}
    m_w &= m_x m_z + m_y r_z + m_z r_x + r_x r_z - r_w \\
    &= m_x m_z + m_y r_z + (m'' - 2\Delta m') + \Gamma r_x + r_x r_z - r_w \\
    &= m_x m_z + m_y r_z + (1 - 2\Delta)(m'r_x + r'') + \Gamma'
\end{align*}
\]
Protocol $\Pi_{\text{ReLU}}(\langle x \rangle)$

Input: $(\cdot)$-shared value of $x$.
Output: $(\cdot)$-shared values of $z = \text{sign}(x)$ and $w = \text{ReLU}(x)$.

**Preprocessing:**
- All parties perform $\langle r'' \rangle$, $\langle r'' \rangle$, $\{r_x, r_w\} \leftarrow \Pi_{\text{ضا}}(\cdot)$;
- $P_i$, for $i \in \{1, 2\}$ pick $\Delta \in \{0, 1\}$ and reveal $[\Gamma'] = \Gamma \cdot \langle r_x \rangle - (1 - 2\Delta)[r''] + [r_x \cdot r_w] - [r_w]$ to each other;
- $P_i$, for $i \in \{1, 2\}$ calculate $[\Gamma'] = \Gamma \cdot \langle r_x \rangle - (1 - 2\Delta)[r''] + [r_x \cdot r_w] - [r_w]$;
- $P_0$ does:
  1. Calculate $r'_x = -r_x - \text{sign}(-r_x) \cdot 2^{\ell-1} \in \mathbb{Z}_{2\ell}$;
  2. Extract $2^{\ell-1} - 1 - r_x$ as $\{r_{x,0}, \ldots, r_{x,\ell-1}\}$;
  3. Perform $\langle r'' \rangle' \leftarrow \Pi_{\text{ضا}}(\langle r_x \rangle)$ for $j \in \mathbb{Z}_{2\ell}$, taking the biggest prime of $p \in \{1, 2^{\ell+1}\}$;
  4. Perform $\langle r_x \cdot r_w \rangle \leftarrow \Pi_{\text{ضا}}(\langle r_x \rangle)$.

**Online:**
- $P_i$, for $i \in \{1, 2\}$ does:
  1. Set $m_i z = m_z - \text{sign}(m_z) \cdot 2^{\ell-1}$ and bitwise it as $\{m_i x_j \in \{0, 1\}\} \in \mathbb{Z}$ while $\sum_{j=0}^{\ell-1} 2^{\ell-1-j} \cdot m_k z \downarrow = m_i z$;
  2. Set $m_j x_j = 0$ and $[r_{x, j}] = [1]$;
  3. Set $[m_{i}'] = m_{i} x_j + [r_{x, j}] - 2 m_{i} x_j \cdot [r_{x, j}]$ for $j \in \mathbb{Z}_{2\ell}$;
  4. Pick same random values $\{w_{i}, w'_{j} \in \mathbb{Z}_{p}\} \in \mathbb{Z}_{2\ell}$ via PRF with seed $\eta$;
  5. Calculate $\{[m_{i}']\} = \sum_{j=0}^{\ell-1} [m_{i}'] = 2 \cdot [m_{i}'] + 1$ and $\{[w_{i}']\} = [w_{i}'] \cdot [m_{i}'] + \text{sign}(m_{i}) \cdot [m_{i} x_j] \cdot \Delta$ and $\{[w'_{j}']\} = [w'_{j}'] \cdot [m_{i}'] + 1$ for $j \in \mathbb{Z}_{2\ell}$;
  6. Pick a random permutation $\pi$ via PRF with seed $\eta$ and permute the list $\{[w_{i}']\} \in \mathbb{Z}_{2\ell}$ and $\{[w'_{j}']\} \in \mathbb{Z}_{2\ell}$ to $P_i$ and reveal $[r'''] = m_{i} x_j \cdot [r_{x, j}]$ to each other in the first round of $\Pi_{\text{ضا}}$.
- $P_0$ sends $m'' = r''$ to $P_i$, for $i \in \{1, 2\}$;
- $P_i$, for $i \in \{1, 2\}$ sets $m_i = m'' - 2\Delta m' + \Gamma + [r''']$ and $m_w = m_w x_i + (1 - 2\Delta)m'' + \Gamma'$;
- All parties output $\langle z \rangle := \langle [r_x], m_z \rangle$ and $\langle w \rangle := \langle [r_w], m_w \rangle$.

**Figure 14:** The 2-round ReLU Protocol.

$m', \Delta, \Gamma$ are the fresh random values mentioned in $\Pi_{\text{ضا}}$ and it hold $m_z = m'' - 2\Delta m' + \Gamma$ in $\Pi_{\text{ضا}}$. We denote $\Gamma' = \Gamma \cdot r_x - (1 - 2\Delta)[r'''] + [r_x \cdot r_w] - [r_w]$ locally in the offline phase. $P_1$ and $P_2$ reveal $[\Gamma'] = \Gamma \cdot \langle r_x \rangle - (1 - 2\Delta)[r'''] + [r_x \cdot r_w] - [r_w]$ locally in the offline phase. $P_1$ and $P_2$ locally calculate $[\Gamma''] = \Gamma \cdot \langle r_x \rangle - (1 - 2\Delta)[r''']$ and $\langle r'' \rangle'$ to $P_1$ and $P_2$. Then $P_1$, $P_2$ locally calculate $m_w = m_w x_i + (1 - 2\Delta)m'' + \Gamma'$ and $m''$. Note that reveal $m''$ and $\Gamma''$ will not leak any information, since the $P_1$ and $P_2$ cannot extract additional information of $r_x, r_w$, besides of $m_w$, with the fresh random value $r''$. Our ReLU protocol requires 1 rounds and communication of $\ell \log \ell + 4\ell$ bits in the preprocessing phase and requires 2 rounds and communication of $4\ell \log \ell + 8\ell$ bits in the online phase.

Figure 15: Evaluate the multiplication one by one, the time’s proportion of the offline, online, and verification phases under the LAN/MAN setting.

**Security.** We analyze the security of our ReLU protocol in the UC framework. The functionality $\mathcal{F}_{\text{ReLU}}$ is defined as an instantiation of $\mathcal{F}_{\text{Sp}}$ depicted in Fig. 2; namely, it calculates $w = \text{ReLU}(x)$ and $z = \text{sign}(x)$.

**Theorem 4.** Let $\text{PRF}_{\mathcal{Z}_p}^{\mathcal{Z}_e}$, $\text{PRF}_{\mathcal{Z}_e}$ and $\text{PRF}_{\mathcal{Z}_f}$ be the secure pseudo-random functions. The protocol $\Pi_{\text{ReLU}}$ depicted in Fig. 8 UC realizes $\mathcal{F}_{\text{ReLU}}$ against semi-honest PPT adversaries who can statically corrupt up to one party.

**Proof.** See Appendix A.7.

**Maxpool.** Our Maxpool scheme is constructed by the comparison function $\text{great}(x, y) = x \geq y$ and the maximum function $\text{max}(x_1, \ldots, x_n)$. In the case of signed numbers $x$ and $y$, $\text{great}(x, y)$ can be implemented by invoking the $\Pi_{\text{ضا}}$ three times. That is, $\text{great}(x, y) = (\text{sign}(x) \oplus \text{sign}(y)) \cdot \text{sign}(y - x) + (1 \oplus \text{sign}(x) \oplus \text{sign}(y)) \cdot \text{sign}(y)$. For unsigned number $x$ and $y$, which $\text{sign}(x) = 0$ and $\text{sign}(y) = 0$, we have $\text{great}(x, y) = \text{sign}(y - x)$. We have observed that after applying Maxpool in the ReLU layer, the sign bit of the data becomes 0. Therefore, we only need to calculate $\text{sign}(y - x)$.

There are two approaches to evaluate $\text{max}(x_1, \ldots, x_n)$. One is to evaluate $\text{max}(x_1, \ldots, x_n)$ by $\text{max}(x_1, \ldots, x_n) = \sum_{i=1}^{n} \langle \langle x_{i}, x_{i+1} \rangle \rangle \cdot \text{great}(x_{i}, x_{i+1} : x_i)$, which perform $\Theta(n^2)$ comparisons in the constant round. The other is to search for the maximum value through the binary tree, i.e. reduce $n$-dimension maximum to $2$-dimension by expanding $\log(n)$ times $\text{max}(x_1, \ldots, x_n) = \text{max}(\text{max}(x_1, x_2), \ldots, x_{n-1}, x_n)$. This method requires $\Theta(\log n)$ rounds to perform a total of $n - 1$ times $2$-dimension maximum.

We observe that the Maxpool procedure may re-use some comparison outcomes more than once while performing the aforementioned maximum operation, depending on the kernel shape and stride. For instance, we assume $\text{z}_{i,j}$ is the result element of performing $(2, 2)$-kernel shape and $1$-stride Maxpool over an $a \times b$-dimension matrix requires where $\text{z}_{i,j} = \text{max}(x_{i,j}, x_{i,j+1}, x_{i+1,j}, x_{i+1,j+1})$ and
While selecting a $W_\text{AN}$. The multiplication depth is logarithmic of the multiplication size.

We implement all the benchmark protocols, we compare the runtime performance with Bicop tor [40], BLAZE [30], SWIFT, and ABY respectively. In this section, we evaluate our multiplication and non-arithmetic protocols in both the semi-honest setting and malicious setting. For the maliciously secure multiplication protocols, we compare the communication and runtime with $\Pi_{\text{Mult}}$ and $\Pi_{\text{Inner}}$ in the Appendix C.1. Fig. 16 depicts the running time of $\Pi_{\text{Mult}}$. We take the different sizes of multiplication with the circuit depth which is equal to the logarithmic of multiplication size. Since the impact of communication on performance is insignificant in the LAN setting, our protocol gathers similar performance to SWIFT in terms of run-time. Meanwhile, influenced by an additional verification round which is the dominant overhead in the case of a small volume of data, our protocol is slightly worse than SWIFT. Considering saturated data, our protocol achieves $2\times$ the performance improvement compared to SWIFT and $10\times$ improvement compared to ABY under both MAN and WAN settings. In Fig. 15, we present a benchmark that showcases the proportion of running time in each phase when executing the multiplication protocol in a one-by-one fashion. This benchmark is conducted after selecting the appropriate value of $R$ using the method previously described. Based on this experiment, we have determined that when the number of rounds required to calculate the multiplication exceeds $2^{10}$, the runtime of our verification phase can be ignored.

**Multiplication.** We compare our maliciously secure multiplication protocol with SOTA. We benchmark the communication of $\Pi_{\text{Mult}}$ and $\Pi_{\text{Inner}}$ in the Appendix C.1. Fig. 16 depicts the running time of $\Pi_{\text{Mult}}$. We take the different sizes of multiplication with the circuit depth which is equal to the logarithmic of multiplication size. Since the impact of communication on performance is insignificant in the LAN setting, our protocol gathers similar performance to SWIFT in terms of run-time. Meanwhile, influenced by an additional verification round which is the dominant overhead in the case of a small volume of data, our protocol is slightly worse than SWIFT. Considering saturated data, our protocol achieves $2\times$ the performance improvement compared to SWIFT and $10\times$ improvement compared to ABY under both MAN and WAN settings. In Fig. 15, we present a benchmark that showcases the proportion of running time in each phase when executing the multiplication protocol in a one-by-one fashion. This benchmark is conducted after selecting the appropriate value of $R$ using the method previously described. Based on this experiment, we have determined that when the number of rounds required to calculate the multiplication exceeds $2^{10}$, the runtime of our verification phase can be ignored.

**Non-arithmetic functions.** The benchmark data in Fig. 17 demonstrates the high efficiency of our nonlinear protocol.

\[ z_{i,j+1} = \max(x_{i,j+1}, x_{i,j+2}, x_{i+1,j+1}, x_{i+1,j+2}). \] Both $z_{i,j}$ and $z_{i,j+1}$ needs the outcome of $\text{great}(x_{i,j+1}, x_{i+1,j+1})$.

We adopt the binary tree solution for its property to eliminate the repeated comparison due to storing the temporary comparison result. The 2-dimension maximum $\max(x_i, x_j)$ can be calculated as $(x_i - x_j) \cdot \text{great}(x_i, x_j) + x_j$, i.e. $(x_i - x_j) \cdot \text{sign}(x_i - x_j) + x_j$. In the previous chapter, we implemented $\hat{f}(x) = x \cdot \text{sign}(x)$ in two rounds by introducing 2ℓ bits of communication overhead in the online phase. We use it to evaluate $\max(x_i, x_j)$ by $\max(x_i, x_j) = x_j - \hat{f}(x_j - x_i)$. We apply this approach to evaluate Maxpool, which requires $(n-1)(\log \ell + 4\ell)$ bits of communication cost in the setup phase and $(n-1)(4\ell \log \ell + 8\ell)$ bits in the online phase.

Analogously, the malicious version of Maxpool can be achieved through verifying sign bit-exact and multiplication respectively.

**7. Implementation and Benchmarks**

In this section, we evaluate our multiplication and non-arithmetic protocols in both the semi-honest setting and malicious setting. For the maliciously secure multiplication protocols, we compare the communication and runtime with SWIFT [24] and ABY [27]. For the non-arithmetic protocols, we compare the runtime performance with Bicop tor [40], BLAZE [30], SWIFT, and ABY respectively.

**Benchmark setting.** We take the size of the ring $\ell = 64$ and the polynomial ring degree $d = 32$. For the fixed-point value, we utilize 16 bits truncation. Our experiments are performed in a local area network, using software to simulate three network settings: local-area network (LAN, RTT: 0.2ms, bandwidth: 1Gbps), metropolitan-area network (MAN, RTT: 12ms, bandwidth: 100Mbps), and wide-area network (WAN, RTT: 160ms, bandwidth: 40Mbps) and executed on a desktop with AMD Ryzen 7 5700X CPU @ 3.4 GHz running Ubuntu 18.04.2 LTS; with 8 CPUs, 32 GB Memory and 1TB SSD.

**Implementation.** We implement all the benchmark protocols based on the Piranha [36] source code [37] which is a GPU platform for MPC protocols. We incorporate the NTL (Number Theory Library) to facilitate calculations involving polynomial rings. We employ the AES encryption of OpenSSL to perform PRG evaluation. For SWIFT and BLAZE, we utilize SHA-256 for its hash function and combine all the messages for each layer into one hash.

![Figure 16: Overall run-time of multiplication. Comparison with ABY3 [27], SWIFT [24] of $\Pi_{\text{Mult}}$ over LAN, MAN and WAN. The multiplication depth is logarithmic of the multiplication size.](image)

**LAN.**

**MAN.**

**WAN.**
We compare the overall running time of the ReLU protocol with SOTA [24], [27], [30], [40] in LAN, MAN, and WAN settings (For Bicoptor, we take the truncation error parameter $\ell^* = 32$). There are little differences in performance between our ReLU protocol and our Maxpool protocol. Owing to page limits, we omit comparative benchmarks of Maxpool against other works in terms of performance. The input size of evaluation is from $2^4$ to $2^{20}$. We perform the protocol 10 times and prepare all random values at once, and finally calculate the amortized run-time. We benchmark our maliciously secure ReLU protocol with different security parameters ($\lambda = 2$ for soundness error $2^{-21}$ and $\lambda = 6$ for soundness error $2^{-48}$). Under the semi-honest threat model and WAN setting, as anticipated, our semi-honest protocol demonstrates a performance improvement of 4× compared to the constant round protocol Bicoptor (theoretically, communication volume has been reduced by 4× on a 64-bit ring). Under the malicious threat model, compared to the constant round protocol BLAZE, our maliciously secure version achieves over 10× performance improvement with a reasonable ReLU size. Since the delay dominates the execution overhead considering the small amount of data, our 2-round protocol is much lower than the logarithmic rounds protocol ABY in terms of time cost. In the above cases, the performance of our protocol is more than 4× that ABY, no matter in LAN, MAN, or WAN settings. The performance of our protocols under a semi-honest setting is provided in Appendix C.2

### The inference of neural network

We further construct the convolutional neural network (CNN) inference. We implement three types of models as follows:

- **Shallow neural network (S-NN).** Our shallow neural network accepts $28 \times 28$ image and involves a convolution layer (5 kernels with $5 \times 5$ shape, the stride of (2,2)), a ReLU layer, and a fully connected layer (connects the incoming $5 \times 13 \times 13$ nodes to the output 10 nodes).
- **LeNet.** We benchmark the LeNet model which replaces the sigmoid activation layer with the ReLU layer. The model accepts $32 \times 32$ image and contains 2-layer convolution, 2-layer Maxpool, 4-layer ReLU and 3-layer full connection.
- **VGG-16.** We benchmark the VGG-16 model which takes $64 \times 64$ image as input and contains 13-layer convolution, 5-layer maxpool, 13-layer ReLU and 8-layer full connection.

### Table 3: Run-time and communication cost of NN inference, under LAN setting.

<table>
<thead>
<tr>
<th>Model</th>
<th>Stage</th>
<th>Offline</th>
<th>Online</th>
<th>Com</th>
<th>Time</th>
<th>Com</th>
<th>Round</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-NN</td>
<td>Execution</td>
<td>0.05</td>
<td>6.07</td>
<td>0.17</td>
<td>2</td>
<td>13.19</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Verification</td>
<td>-</td>
<td>-</td>
<td>1.75</td>
<td>3</td>
<td>23.52</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LeNet</td>
<td>Execution</td>
<td>0.65</td>
<td>7.40</td>
<td>2.46</td>
<td>42</td>
<td>104.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Verification</td>
<td>-</td>
<td>-</td>
<td>26.1</td>
<td>10</td>
<td>118.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VGG</td>
<td>Execution</td>
<td>10.2</td>
<td>207</td>
<td>39.2</td>
<td>127</td>
<td>8341</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Verification</td>
<td>-</td>
<td>-</td>
<td>414</td>
<td>18</td>
<td>12157</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FIGURE 17: Overall run-time of ReLU in LAN/MAN/WAN setting. Where ours-semi refers to our semi-honestly secure protocol; ours-mal refers to our maliciously secure protocols (Soundness error $2^{-48}$ for $\lambda = 6$ and $2^{-21}$ for $\lambda = 2$ ); Bicoptor refers to [40]; BLAZE refers to [30]; Swift refers to [24]; ABY refers to [27].
ReLU, and MaxPool. The experiments show that our various protocols have significant performance improvements over the state-of-the-art works, i.e., [24], [27], [30], [40].

References


[34] Sameer Wagh, Divya Gupta, and Nishanth Chandran. Securenn: 3-party secure computation for neural network training. In PoPETs, 2019.


[40] Lijing Zhou, Ziyu Wang, Hongrui Cui, Qingrui Song, and Yu Yu. Bi-pseudo-random functions. The protocol

Appendix A.

Security Proofs

A.1. The proof of Theorem 1.

Theorem 1. Let $\text{PRF}_{\text{prf}}^{\text{PRF}_{\text{prf}}^{\text{PRF}_{\text{prf}}}}$ be the secure pseudo-random functions. The protocol $\Pi_{\text{SignBit}}$ as depicted
in Fig. 3 UC realizes $F_{\text{SignBit}}$ against semi-honest PPT adversaries who can statically corrupt up to one party.

Proof. To prove Thm. 1, we construct a PPT simulator $S$, such that no non-uniform PPT environment $Z$ can distinguish between the ideal world and the real world. We consider the following cases:

Case 1: $P_0$ is corrupted.

Simulator: The simulator $S$ internally runs $A$, forwarding messages to/from $Z$ and simulates the interface of honest $P_1$, $P_2$. $S$ simulates the following interactions with $A$.

- Upon receiving $\{[r_{x,j}]_1\}_{j \in \mathbb{Z}_p}$, $(r_{x,j})_2$ form corrupted $P_0$ to $P_1$, and $\{[r_{x,j}]_2\}_{j \in \mathbb{Z}_p}$ form corrupted $P_0$ to $P_2$, $S$ extracts $r_x = 2^{-1} - 1 - \sum_{j=1}^{\ell-1}([r_{x,j}]_1 + [r_{x,j}]_2)$ and $r' = [r']_1 + [r']_2$.

- $S$ picks random list $\{u_j\}_{j \in \mathbb{Z}_p}$, where $u_j \in \mathbb{Z}_p$ and sets another list $\{u_j\}_{j \in \mathbb{Z}_p}$ as follows:

  - For each $j$ where $u_j' = 0$, set $u_j \leftarrow \{p - 1, 0\}$.
  - For each $j$ where $u_j' \neq 0$, set $u_j \leftarrow \mathbb{Z}_p$.
  - If $\exists j$ such that $u_j' \neq 0$, select random one of $j$ to set $u_j \leftarrow \mathbb{Z}_p$.
  - $S$ sends $\{u_j\}_j \in \mathbb{Z}_p$ and $\{u_j\}_j \in \mathbb{Z}_p$ to $P_0$.

- Upon receiving $m'$ from corrupted $P_0$ to $P_1$ and $P_2$, $S$ does:

  - If $\exists u_j = 0 \wedge u_j' \neq 0$, set $\text{sign}(r_x) = (m'+r')$, else set $\text{sign}(r_x) = (m'+r') \oplus 1$.
  - Calculate $r_x = r_x - \text{sign}(r_x) \cdot 2^{\ell-1}$.
  - $S$ sends $(\text{Input}, r_x) \rightarrow F_{\text{SignBit}}$.

Indistinguishability. The indistinguishability is proven through a series of hybrid worlds $H_0, H_1$.

Hybrid $H_0$: It is the real protocol execution $\text{Real}_{\text{SignBit}, A, Z}(\ell, \lambda)$.

Hybrid $H_1$: It is same as $H_0$ except that in $H_1$, list $u_j$ and $u_j'$ are picked uniformly random instead of calculating from $u_j \cdot m_j + (\text{sign}(m_i) \oplus \hat{m}_{z_{i,j}} + \Delta)$ and $x_j' \cdot (w_j \cdot x_j + 1)$.

Claim 1. If $\text{PRF}^e_{\ell, \lambda}$ and $\text{PRF}^e_{\ell, \lambda}$ are the secure pseudorandom functions with adversarial advantage $\text{Adv}_{\text{PRF}^e_{\ell, \lambda}}$, then $H_1$ and $H_0$ are indistinguishable with advantage $\epsilon < 2 \cdot \ell \cdot \text{Adv}_{\text{PRF}^e_{\ell, \lambda}}$.

Case 2: $P_1$ (or $P_2$) is corrupted.

Simulator: The simulator $S$ internally runs $A$, forwarding messages to/from $Z$ and simulates the interface of honest $P_0$, $P_2$. $S$ simulates the following interactions with $A$.

- $S$ generates $\Delta$, $[r_2]_1$ with the seed and sends $[r_2]_1$ to $F_{\text{SignBit}}$.

- $S$ picks $[r_{x,j}]_1 \leftarrow \mathbb{Z}_p$ and acts as $P_0$ to send it to $P_1$.

- $S$ picks $[\Gamma]_2 \leftarrow \mathbb{Z}_p$ and acts as $P_2$ to send it to $P_1$.

- $S$ generates $\Gamma$ from $P_1$. $S$ calculates $\Gamma = [\Gamma]_1 + [\Gamma]_2$.

- Upon receiving $\{[u_j]\}_{j \in \mathbb{Z}_p}$ and $\{[u_j]\}_{j \in \mathbb{Z}_p}$ from corrupted $P_1$ to $P_0$, $S$ does:

  - invoke PRF with $\eta$ to generate permutation $\pi$.
  - invoke PRF with $\eta$ to generate $w_j, w_j' \in \mathbb{Z}_p$ for $j \in \mathbb{Z}_p$.
  - invoke PRF with $\eta$ to generate $\Delta \in \mathbb{Z}_p$.
  - calculate $\{[u_j]\}_{j \in \mathbb{Z}_p}$ via $\{[u_j]\}_{j \in \mathbb{Z}_p}$ and $\{[u_j]\}_{j \in \mathbb{Z}_p}$ and $\{[w_j]\}_{j \in \mathbb{Z}_p}$.

- $S$ sends $(\text{Input}, m) \rightarrow F_{\text{SignBit}}$ and receives $(\text{Output}, m, [r_2]_1)$.

- $S$ acts as $P_0$ to send $m' = (m_z - \Delta) / (1 - 2\Delta)$ to $P_1$.

Indistinguishability. The indistinguishability is proven through a series of hybrid worlds $H_0, H_1$.

Hybrid $H_0$: It is the real protocol execution $\text{Real}_{\text{SignBit}, A, Z}(\ell, \lambda)$.

Hybrid $H_1$: It is same as $H_0$ except that in $H_1$, $[r_{x,j}][\Gamma]_1$ and $m'$ are picked uniformly random instead of calculating from $r_{x,j}$, $\Delta + [r']_2 - 2\Delta \cdot [r']_2 + [r_2]_2$ and $\text{sign}(-r_x) - r'$.

Claim 2. If $\text{PRF}^e_{\ell, \lambda}$ and $\text{PRF}^e_{\ell, \lambda}$ are the secure pseudorandom functions with adversarial advantage $\text{Adv}_{\text{PRF}^e_{\ell, \lambda}}$ and $\text{Adv}_{\text{PRF}^e_{\ell, \lambda}}$, then $H_1$ and $H_0$ are indistinguishable with advantage $\epsilon = \ell \cdot \text{Adv}_{\text{PRF}^e_{\ell, \lambda}} + 2\text{Adv}_{\text{PRF}^e_{\ell, \lambda}}$.

Proof. We replace the $\ell$ $\text{PRF}^e_{\ell, \lambda}$ outputs and $2\text{PRF}^e_{\ell, \lambda}$ outputs to uniformly random number; therefore, the overall advantage is $\epsilon = \ell \cdot \text{Adv}_{\text{PRF}^e_{\ell, \lambda}} + 2\text{Adv}_{\text{PRF}^e_{\ell, \lambda}}$ by hybrid argument via reduction.

This concludes the proof.

A.2. The proof of Lemma 1.

Lemma 1. Suppose protocol $\Pi_{\text{Trans}}$ depicted in Fig. 4 take input as $\{(x_i, y_i), (z_i)\}_{i \in \mathbb{Z}_N}$, and it outputs $\{(x_i', y_i', z_i')\}_{i \in \mathbb{Z}_N}$; $(z_i')$. The probability that the following two conditions hold is at most $\frac{N}{2}$, where $d$ is the degree of $f(x)$ w.r.t. $\mathbb{Z}_2[x]/f(x)$:

- $z = \sum_{i=0}^{\mathbb{N}} x_i'$, $y_i'$
- $\exists i \in \mathbb{Z}_N$ s.t. $z_i' \neq x_i \cdot y_i$

Proof. It is sufficient to show that $r$ is uniformly random if $\Pi_{\text{sec}}$ is not abort. The adversary tries to make $\sum_{i=0}^{\mathbb{N}} r_i$. $z_i' = \sum_{i=0}^{\mathbb{N}} r_{i-1} \cdot x_i \cdot y_i$ where $z_i = x_i \cdot y_i$, $e_i$ for $i \in \mathbb{Z}_N$, with an error list $\{e_i\}_{i \in \mathbb{Z}_N}$. It can be written as $\sum_{i=0}^{\mathbb{N}} r_i$. $x_i \cdot y_i = \sum_{i=0}^{\mathbb{N}} r_{i-1} \cdot x_i \cdot y_i + e_i$. The condition that makes the equation hold is the random value $r$ is the root of $f(x) = \sum_{i=0}^{\mathbb{N}} x_i e_i$. Since the size of roots of $N - 1$-degree $f(x)$ over $\mathbb{Z}_2[x]$ is $2^{(\ell-1)d}N + 1$, the probability that uniformly random value $r$ match the root is $\frac{2^{(\ell-1)d}N + 1}{2^{\ell d}} \approx \frac{N}{2}$. □
A.3. The proof of Lemma 2.

Lemma 2. Suppose protocol $\Pi_{Reduct}$ depicted in Fig. 5.

take input as $\{(x_i)^{(l,x)}, (y_i)^{(l,x)}\}_{i \in \mathbb{Z}_N}, (z)^{(l,x)}$, and it outputs $\{(x_i)^{(l)}, (y_i)^{(l)}\}_{i \in \mathbb{Z}_N\setminus \{1\}}, (z)^{(l)}$. The probability that the following two conditions hold is at most $\frac{1}{2^\frac{r}{\ell + \epsilon}}$, where $d$ is the degree of $f(x)$ w.r.t. $\mathbb{Z}_{2^d}[x]/f(x)$:

- $z' = \sum_{i=0}^{N/2} x_i \cdot y_i$
- $z \neq \sum_{i=0}^{N-1} x_i \cdot y_i$

Proof. For the convenience of description, we denote $h'(k) = \sum_{j=0}^{N/2} f_i(k) \cdot y_i$. The adversary tries to make $h(k) = h'(k) + e$ when $h(0) + h(1) = h'(0) + h'(1) + e$ (we denote $e$ the error introduced in $z$). At the same time, the adversary can make new errors $e_1,e_2$ when calculating $h(0)$ and $h(2)$ so that $h(0) = h'(0) + e_1,h(1) = h'(1) + e - e_1,h(2) = h'(2) + e_2$. Considering $h(k) = \sum_{i=0}^{N-1}[[z_i = 0]](f_i(k) - f_i'(k)) \cdot h(i)$, to make it equal to $h'(k) = (\sum_{i=0}^{N-1} f_i'(k) \cdot h(i)) + \sum_{i=0}^{N-1} f_i'(k) \cdot h(i) = \frac{(\sum_{i=0}^{N-1} f_i'(k) \cdot h(i)) + 2\sum_{i=0}^{N-1} f_i'(k) \cdot h(i)}{2}$, is to make $2\sum_{i=0}^{N-1} f_i'(k) \cdot h(i) = \sum_{i=0}^{N-1} f_i'(k) \cdot h(i)$, which is at most $2^2\sum_{i=0}^{N-1} f_i'(k) \cdot h(i)$, so we have the soundness error $2^{2\sum_{i=0}^{N-1} f_i'(k) \cdot h(i)} = \frac{1}{2^{\frac{r}{\ell + \epsilon}}}$. □

A.4. The proof of Lemma 3.

Lemma 3. Let $\{(x_i)^{(l)}, (y_i)^{(l)}\}_{i \in \mathbb{Z}_N}, (z)^{(l)}$ be the input of protocol $\Pi_{InnerVerify}$ depicted in Fig. 6. The probability that $\Pi_{InnerVerify}$ outputs 1 and $z \neq \sum_{i=0}^{N} x_i \cdot y_i$ is at most $\frac{1}{2^d}$, where $d$ is the degree of $f(x)$ w.r.t. $\mathbb{Z}_{2^d}[x]/f(x)$.

Proof. Since $\alpha$ is uniformly random and unknown to the adversary, for $z = \sum_{i=0}^{N} x_i \cdot y_i + e$, we have $\Delta = \alpha \cdot e_1$ where $e_1$ is introduced when evaluating $\Pi_{PolyEval}$. Since $\Pi_{PolyEval}$ is secure up to additive attack, $e_1$ is independent of $\alpha$, so that polynomial $f(x) = e \cdot x + e_1$ over $\mathbb{Z}_{2^d}[x]$ has $2^d/2^d + 1$ roots. The probability the adversary deliberately chooses $e,e_1$ to make $\Delta = 0$ is $2^{\frac{d-1}{2^d} - 1} \approx \frac{1}{2^d}$. □

A.5. The proof of Theorem 2.

Theorem 2. Let $\{(x_i), (y_i), (z_i)\}_{i \in \mathbb{Z}_N}$ be the input of protocol $\Pi_{MultiVerify}$ depicted in Fig. 7. The probability $\Pi_{MultiVerify}$ outputs 1 and $\exists i \in \mathbb{Z}_N\ s.t. z_i \neq x_i \cdot y_i$ is at most $\frac{1}{2^d}$, where $d$ is the degree of $f(x)$ w.r.t. $\mathbb{Z}_{2^d}[x]/f(x)$.

Proof. From Lemma 1, Lemma 2 and Lemma 3, we know that the adversary has $R$ chances with probability $\frac{1}{2^d}$ and one chance with probability $\frac{1}{2^d}$, and one chance with probability $\frac{1}{2^d}$ to pass the verification. Therefore the probability that the adversary success is $1 - (1 - \frac{1}{2^d})^R \cdot (1 - \frac{1}{2^d}) \cdot (1 - \frac{1}{2^d}) \approx \frac{N}{2^d-n-2}^R$. □

A.6. The proof of Theorem 3.

Theorem 3. Let $\langle x \rangle^l$ be the input of the protocol $\Pi_{Pos}^\lambda$ depicted Fig. 8. The probability that $\Pi_{Pos}^\lambda$ outputs 1 and $\text{sign}(x) = 1$ is at most $\frac{1}{2^\frac{\log m + \ell + \epsilon}{2}}$.

Proof. For each illegal $u_i$ in $\Pi_{Pos}^\lambda$, the probability the malicious $P_i$ for $i \in \{1,2\}$ make it pass the MAC check is $\frac{1}{2^\frac{\log m + \ell + \epsilon}{2}}$ w.r.t. the MAC key space $\mathbb{Z}_p$ (taking $p \approx 2^{(\log m + \ell + \epsilon)}$). To persuade the verifier to accept the result, the adversary also needs to guess the position of the first non-zero bit and flip the coin with probability $\frac{1}{2}$. So the soundness error is $\frac{1}{2^\frac{\log m + \ell + \epsilon}{2}} = \frac{1}{2^\frac{\log m + \ell + \epsilon}{2}}$. □

A.7. The proof of Theorem 4.

Theorem 4. Let $\text{PRF}_{\text{Key},\lambda}^\ell, \text{PRF}_{\text{Key}}^\ell$ and $\text{PRF}_{\text{Key}}^\ell$ be the pseudo-random functions. Let $\ell = \text{poly}(\lambda)$. The protocol $\Pi_{Secure}$ depicted in Fig. 8 UC realizes $F_{\text{SigBit}}$ against semi-honest PPT adversaries who can statically corrupt up to one party.

Proof. The proof of Theorem 4 is similar to the proof of Theorem 1. To be brief, compared to $\Pi_{Secure}$, $\Pi_{Relu}$ sends $m''$ to corrupted $P_i$ (or $P_2$) and introduce the PRF outputs $x'$, $r'$ to make $m''$ uniformly random, where make the indistinguishable advantage of corrupted $P_i$ equal to $\epsilon = \ell \cdot \text{Adv}_{\text{PRF}_{\text{Key}}^\ell}(1^\lambda,A) + 2\text{Adv}_{\text{PRF}_{\text{Key}}^\ell}(1^\lambda,A)$ □

Appendix B.

Other semi-honest protocol

B.1. Semi-honest secure truncation

Our truncation protocol $\Pi_{semi-trunc}$ under the semi-honest threat model is shown in Fig. 18, which only requires one round and communication of $\ell$ bits in the offline phase.

![Figure 18: The semi-honest truncation protocol](image-url)
Figure 19: Communication overhead comparison with ABY3 [27], BLAZE [30], SWIFT [24] of multiplication and inner product.

**Appendix C. Benchmarks**

**C.1. Multiplication communication comparison**

Fig. 19 shows our communication overhead compared with ABY, BLAZE, and SWIFT. We take the vector dimension 1024 when evaluating the inner product. Since our protocol requires logarithmic additional communication of \((10R + 8)\ell \cdot d\) (take \(R = \log N\)), it requires more communication than SWIFT given the small \(N\). When \(N\) is large enough, the logarithmic scaler \(R\) makes the additional term ignorable. With a considerable amount of input size, the increase in communication volume of our protocol is \(2\times\) of SWIFT and \(7\times\) of ABY for multiplication and \(2\times\) of SWIFT and \(7168\times\) of ABY for the 1024-dimension inner product with truncation.

**C.2. Non-arithmetic protocol benchmark in semi-honest setting**

Our non-arithmetic protocol benchmark in the semi-honest setting is illustrated in TABLE 4.

**C.3. The communication of our protocols**

We summarize the overhead of our protocols of Multiplication, Inner Product, Truncation, Sign-bit Extraction, ReLU, and MaxPool which is depicted in TABLE 5.

**C.4. Determination of \(R\)**

We determine the \(R\) of multiplication verification in different environments through experiments. The data presented in Fig. 20 illustrates the time required for different numbers of triples for verification using two methods: the scheme that involves calling \(\Pi_{\text{Reduce}}\) to reduce dimensionality to 1, and the scheme that directly employs \(\Pi_{\text{InnerVerify}}\). These measurements were conducted in three distinct network environments. To determine the optimal value of \(R\), the process involved finding the point of intersection \(P\) between the gradients of the two schemes in the figure. Once \(P\) is identified, the dimension is reduced to a value less than \(P\) using the \(\Pi_{\text{Reduce}}\) operation performed minimum \(R\) times (As shown in Fig. 20, we take \(P = 2^{11}\) in LAN, \(P = 2^{15}\) in MAN and \(P = 2^{16}\) in WAN).

**Appendix D. Related work**

In the honest-majority setting, several works such as [11], [13], [15], [16], [18], [39] have designed protocols for efficient secure multi-party computation against the malicious adversary. However, compared to the semi-honest case, previous work requires significantly higher additional overhead. For instance, [39] presents two sets of schemes that require a communication overhead of either \(42 \cdot n\) or \(5(n^2 - n)\) ring elements for each multiplication, where \(n\) represents the number of parties. [11] reduces the communication overhead to \(42 \cdot n\). [18] introduces batch verification and a series of other optimization techniques. These protocols by [18] require a two-round communication overhead of \(2n\) field elements or a one-round communication overhead of \(3n\) field elements. However, it should be noted that [18]’s protocol can only run on the field. In contrast, [16] achieves a constant online phase communication overhead of 12 field elements by utilizing packed secret sharing technology. Lastly, the work by [13] refocuses on secure multi-party computation in a ring setting. It achieves a communication overhead of \(1\frac{1}{2}n\) ring elements with two rounds of communication or \(1\frac{1}{4}n\) ring elements with one round of communication. With the advancement of the maliciously secure multiplication protocol, practical maliciously secure privacy-preserving machine learning becomes attainable. [6], [8], [9], [24], [27], [30], [31], [35] realize privacy-preserving machine learning protocols under the malicious threat model in an honest majority. In the semi-honest setting, protocols such as [8], [27], [29], [30] are all based on three parties replicated secret sharing, which only request 3 ring elements communication each multiplication. The online phase communication overhead of 2 ring elements can be achieved by handing over part of the communication to a circuit-dependent offline phase [8]. In the malicious setting, different from the overhead of 21 ring elements (12 in the offline phase) [27], a series of optimizations [8], [24], [30] reduced the multiplication overhead to 6 ring elements (3 in the offline phase) in the three-party setting. To evaluate non-linear functions such as ReLU and Maxpool, protocols like [24], [27], [29] employ a conversion process that transforms arithmetic secret sharing into boolean secret sharing. Subsequently, they utilize this boolean secret-sharing scheme to evaluate corresponding non-linear functions. The disadvantage of this approach is the need to introduce \(\log \ell\) rounds.
TABLE 4: Runtime and communication cost of each non-arithmetic protocol evaluation in semi-honest, MAN setting.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Input Size</th>
<th>Communication</th>
<th>Time (ms)</th>
<th>Throughput (ops/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Offline</td>
<td>Online</td>
<td>Offline</td>
</tr>
<tr>
<td>Sign</td>
<td>$2^4$</td>
<td>1.1 KB</td>
<td>4.2 KB</td>
<td>11.52</td>
</tr>
<tr>
<td></td>
<td>$2^8$</td>
<td>16.6 KB</td>
<td>66.4 KB</td>
<td>11.96</td>
</tr>
<tr>
<td></td>
<td>$2^{16}$</td>
<td>4.2 MB</td>
<td>17.0 MB</td>
<td>77.59</td>
</tr>
<tr>
<td>ReLU</td>
<td>$2^4$</td>
<td>1.3KB</td>
<td>5.2KB</td>
<td>11.67</td>
</tr>
<tr>
<td></td>
<td>$2^8$</td>
<td>20.7KB</td>
<td>83.1KB</td>
<td>11.96</td>
</tr>
<tr>
<td></td>
<td>$2^{16}$</td>
<td>5.3MB</td>
<td>21.2MB</td>
<td>77.71</td>
</tr>
<tr>
<td>MaxPool</td>
<td>$2^4$</td>
<td>1.1KB</td>
<td>5.1KB</td>
<td>11.75</td>
</tr>
<tr>
<td></td>
<td>$2^8$</td>
<td>20.6KB</td>
<td>82.8KB</td>
<td>11.86</td>
</tr>
<tr>
<td></td>
<td>$2^{16}$</td>
<td>5.3MB</td>
<td>21.2MB</td>
<td>76.04</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Operation</th>
<th>Execution (Semi-honest)</th>
<th>Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplication</td>
<td>$\ell$</td>
<td>1</td>
</tr>
<tr>
<td>Inner Product</td>
<td>$\ell$</td>
<td>1</td>
</tr>
<tr>
<td>Truncation</td>
<td>$\ell$</td>
<td>0</td>
</tr>
<tr>
<td>Sign-bit Extraction</td>
<td>$\ell(1 + \log \ell) + 2\ell$</td>
<td>2</td>
</tr>
<tr>
<td>ReLU</td>
<td>$\ell(1 + \log \ell) + 3\ell$</td>
<td>2</td>
</tr>
<tr>
<td>MaxPool</td>
<td>$(n-1)(4\ell + \ell \log \ell)$</td>
<td>$\log n$</td>
</tr>
</tbody>
</table>

Figure 20: Total runtime comparison of two types batch verification over LAN(0.1ms RTT and 1Gbps bandwidth), MAN(6ms RTT and 100Mbps bandwidth) and MAN(80ms RTT and 40Mbps bandwidth). Of communication. Furthermore, in protocols such as [8], [27], [30], garbled circuits are employed for evaluating non-linear functions. While these protocols exhibit a constant number of communication rounds, the use of garbled circuits introduces a significant amount of additional communication overhead, particularly in the presence of a malicious threat model. In contrast, the protocols described in [25], [34] tackle the sign-bit extraction problem with a constant round communication overhead. They achieve this by converting the highest bit problem into the least significant bit problem. However, when evaluating protocols such as ReLU, they require a substantial communication overhead of 10 rounds, which can be even larger than $\log \ell$ rounds when $\ell$ is small. On the other hand, [40] implements comparison through a truncation protocol. Their approach performs local truncation $\ell$ times, followed by involving a third party to verify if the result contains zero items. This scheme realizes two rounds of $\ell^2$ bits communication. However, this approach has not been applied to malicious threat models.