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: {lutianpei, bingsheng, kuiren}@zju.edu.cn.
†Ant Group Co.,Ltd, Hangzhou, China, Email: lichun.llc@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 3ℓ 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], [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 IT-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

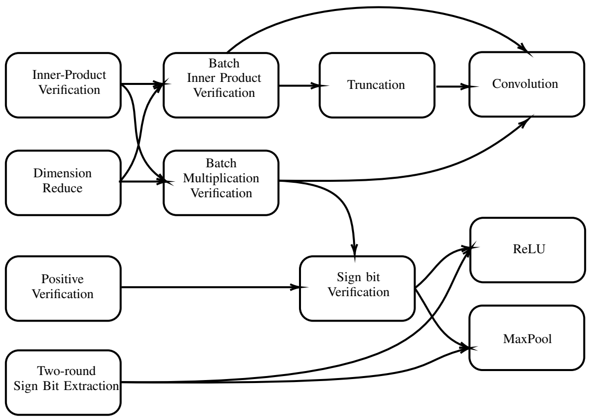


Figure 1: The roadmap of Aegis

reduction technique [19], which can half the inner product vector dimension per iteration with a constant overhead. (cf. Sec. 5.1).

Next, we propose a novel secure comparison (a.k.a. sign bit extraction) protocol Π_{SignBit}. The intuition of our secure comparison is to transfer the sign bit extraction problem to checking whether a certain coordinate of a list is 0. More specifically, our protocol lets two parties generate such a transferred list, and the other party perform the zero check (cf. Sec. 4). The procedure requires two rounds of communication in the online phase.

We further design the maliciously secure verification protocol Π_{VSignBit} to audit the correctness of our secure comparison protocol. Our main observation is that the underlying replicated share is symmetric, and there is at most one malicious party among the 3 MPC participants. Following the dual execution paradigm [23], we perform the check twice. For each check, we nominate a different party to play the role of the verifier and let him generate IT-secure MAC to the share and check the execution correctness. The comparison result shall be accepted if and only if both verifications pass (cf. Sec. 5.2).

As shown in Fig. 1, we first design a maliciously secure inner product verification protocol $\Pi_{\mathsf{InnerVerify}}$ that can check the correctness of an inner product gate. We then adapt the maliciously secure dimension reduction protocol Π_{Reduce} to the ring setting as mentioned before. After that, we propose a maliciously secure positive assertion protocol Π_{Pos} that can assert a shared value is positive, i.e., the sign bit is 0.

Our batch multiplication verification protocol $\Pi_{\mathsf{MultVerify}}$ and inner product verification protocol $\Pi_{\mathsf{InnerVerify}}$ serve the purpose of batch-verifying the correctness of multiplication triples and inner product triples. These protocols are built on top of the aforementioned $\Pi_{\mathsf{InnerVerify}}$ and Π_{Reduce}. Additionally, we also develop a maliciously secure truncation protocol Π_{Trunc} with no online communication. Finally, we built the convolution protocol, the ReLU protocol Π_{ReLU} 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, 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 CrypTFlow [25] (8-round with $6\ell log\ell + 14\ell$ bits communication) and Bicoptor [40] (2-round with the $(\ell^* + \ell)(2 + \ell)$ bits communication, with error probability $2^{1-\ell^*}$), our protocol demonstrates significant improvements (2-round with $4\ell \log \ell + 6\ell$ communication). Specifically, our protocol reduces the communication cost by 75% for the semi-honest setting. Furthermore, in real-world benchmark tests, our protocol exhibits $4\times$ 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 $10\lambda\ell(\log \ell + 1) + 14\ell \log \ell + 16\ell$ bits communication while λ is the statistical security parameter and the soundness error is $2^{-(\lambda \log \ell + \lambda + \log \ell)}$. To the best of our knowledge, our maliciously secure protocol significantly reduces communication of SOTA constant round solutions. Compared with BLAZE [30] (5-rounds with $5\kappa\ell + 6\ell + \kappa$ bits communication in the offline phase and 4-round and $\kappa\ell + 6\ell$ bits communication in the online phase), our protocol reduces both the communication and round complexity by 50%, when $\ell = 64, \kappa = 128$ and $\lambda = 6$ (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 $\mathbb{Z}_{2^\ell}$ against malicious adversaries typically incurs a higher overhead. In this work, we propose a new maliciously secure 3PC multiplication protocol over ring $\mathbb{Z}_{2^\ell}$ 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 multiplication over ring [13], our protocol reduces the overall communication by 40%. Note that [13] achieves full security in the $\mathcal{Q}^3$ active adversary setting ($t < n/3$), while our protocol achieves security with abort in the $\mathcal{Q}^2$ 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

TABLE 1: Comparison of 3-PC based PPML. (ℓ is the ring size, ℓ^* is the security parameter for truncation error $2^{1-\ell^*}$, n is the size of the inner product, $\kappa = 128$ is the security parameter of GC, and $\lambda = 5$ is the statistical security parameter.)

Operation	Protocol	Offline	Online		Malicious
		Communication (bits)	Rounds	Communication (bits)	
Mult	ABY3 [27]	12ℓ	1	9ℓ	✓
	BLAZE [30]	3ℓ	1	3ℓ	✓
	SWIFT [24]	3ℓ	1	3ℓ	✓
	Ours	1ℓ	1	2ℓ	✓
Inner Product	ABY3 [27]	$12n\ell$	1	$9n\ell$	✓
	BLAZE [30]	$3n\ell$	1	3ℓ	✓
	SWIFT [24]	3ℓ	1	3ℓ	✓
	Ours	1ℓ	1	2ℓ	✓
Inner Product with Trunction	ABY3 [27]	$12n\ell + 84\ell$	1	$9n\ell + 3\ell$	✓
	BLAZE [30]	$3n\ell + 2\ell$	1	3ℓ	✓
	SWIFT [24]	15ℓ	1	3ℓ	✓
	Ours	7ℓ	1	2ℓ	✓
ReLU	ABY3 [27]	60ℓ	$3 + \log \ell$	45ℓ	✓
	BLAZE [30]	$5\kappa\ell + 6\ell + \kappa$	4	$\kappa\ell + 6\ell$	✓
	SWIFT [24]	21ℓ	$3 + \log \ell$	16ℓ	✓
	Falcon [35]	0	$5 + \log \ell$	32ℓ	✓
	Bicoptor [40]	0	2	$(\ell^* + \ell)(2 + \ell)$	×
	Ours (Semi-honest)	$\ell \log \ell + 4\ell$	2	$4\ell \log \ell + 8\ell$	×
	Ours (Malicious)	$\ell \log \ell + 4\ell$	2	$10\lambda\ell(\log \ell + 1) + 14\ell \log \ell + 16\ell$	✓

50% of that in SWIFT [24].

2. Preliminaries

Notation. Let $\mathcal{P} := \{P_0, P_1, P_2\}$ be the three MPC parties. During the PPML 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 $\lfloor x \cdot 2^k \rfloor$; if $x < 0$, we encode it as $2^\ell + \lfloor x \cdot 2^k \rfloor$. 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 $\gamma(x) = \alpha \cdot x$ as the MAC of x where α is the MAC key. We denote $\mathsf{sign}(x)$ as the sign bit of x and $\mathsf{rshift}(x)$ 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 $\mathbb{Z}_{2^\ell}$ as $[x] := ([x]_1 \in \mathbb{Z}_{2^\ell}, [x]_2 \in \mathbb{Z}_{2^\ell})$ 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 $\mathbb{Z}_{2^\ell}$ as $\langle x \rangle := ([r_x], m_x \in \mathbb{Z}_{2^\ell})$ where r_x is a fresh random value and $m_x = r_x + x$. P_i for $i \in \{1, 2\}$ hold $(m_x \in \mathbb{Z}_{2^\ell}, [r_x]_i \in \mathbb{Z}_{2^\ell})$ and P_0 holds $([r_x]_1, [r_x]_2)$.
- $[\![\cdot]\!]^p$-sharing: We define $[\![\cdot]\!]^p$ over finite field $\mathbb{Z}_p$ as $[\![x]\!]^p := ([\![x]\!]_1 \in \mathbb{Z}_p, [\![x]\!]_2 \in \mathbb{Z}_p)$ where $x = [\![x]\!]_1 + [\![x]\!]_2 \pmod p$. P_i for $i \in \{1, 2\}$ hold share $[\![x]\!]_i$ and P_0 holds the plaintext x.
- $\| \cdot \|_i^{p,\lambda}$-sharing: We define $\| \cdot \|_i^{p,\lambda}$-sharing over finite field $\mathbb{Z}_p$ as $\|x\|^{p,\lambda} := ([\![x]\!]^p, \{[\![\alpha_j]\!]^p, [\![\gamma(x)_j]\!]^p\}_{j \in \mathbb{Z}_\lambda})$. In our sign-bit verification protocol, one party P_i holds the plaintext of $(x, \{\alpha_j, \gamma(x)_j\}_{j \in \mathbb{Z}_\lambda})$, and the other parties P_k for $k \in \{i - 1 \pmod 3, i + 1 \pmod 3\}$ hold the share $([\![x]\!]_k, \{[\![\alpha_j]\!]_k, [\![\gamma(x)_j]\!]_k\}_{j \in \mathbb{Z}_\lambda})$.

We use $[\cdot]^{\ell[x]}$ and $\langle \cdot \rangle^{\ell[x]}$ to denote the share in the polynomial ring $\mathbb{Z}_{2^\ell}[x]/f(x)$ where $f(x)$ is a degree-d irreducible polynomial over $\mathbb{Z}_2$. For $\| \cdot \|_i^{p,\lambda}$ we utilize subscript i to denote that the plaintext is held by P_i. Note that we let any two shared values $\|x\|_i^{p,\lambda}$ and $\|y\|_i^{p,\lambda}$ for plaintext holder P_i use the same MAC key. For simplicity, we use $\| \cdot \|$, $[\![\cdot]\!]$ when semantics are clear.

All the aforementioned secret-sharing forms have the linear homomorphic property, i.e., $[x] + [y] = ([x]_1 + [y]_1, [x]_2 + [y]_2)$ and $c \cdot [x] = (c \cdot [x]_1, c \cdot [x]_2)$ and $[x] + c = ([x]_1 + c, [x]_2)$, where c is a public value. The same linear operation holds for $\langle \cdot \rangle$, $[\![\cdot]\!]$, and $[\cdot]^{\ell[x]}, \langle \cdot \rangle^{\ell[x]}$. For $\| \cdot \|$, we have $\|x\| + \|y\| = ([\![x]\!] + [\![y]\!], \{[\![\alpha_j]\!], [\![\gamma(x)_j]\!] + [\![\gamma(y)_j]\!]\}_{j \in \mathbb{Z}_\lambda})$, $c \cdot \|x\| = (c \cdot [\![x]\!], \{[\![\alpha_j]\!], c \cdot [\![\gamma(x)_j]\!]\}_{j \in \mathbb{Z}_\lambda})$ and $c + \|x\| = (c + [\![x]\!], \{[\![\alpha_j]\!], c \cdot [\![\alpha_j]\!] + [\![\gamma(x)_j]\!]\}_{j \in \mathbb{Z}_\lambda})$.

Secret sharing. Let $\Pi_{[\cdot]}$, $\Pi_{[\![\cdot]\!]}$, $\Pi_{\langle \cdot \rangle}$, and $\Pi_{\| \cdot \|}$ to denote the corresponding secret sharing protocols. By $\Pi_{[\cdot]}(x)$, we mean that x is shared by P_0; by $\Pi_{[\cdot]}$, 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.

- $[x] \leftarrow \Pi_{[\cdot]}(x)$:
  - P_0 and P_1 pick random value $[x]_1 \in \mathbb{Z}_{2^\ell}$;
  - P_0 sends $x_2 = x - [x]_1 \pmod{2^\ell}$ to P_2.
- $[x] \leftarrow \Pi_{[\cdot]}$:
  - P_0 and P_1 pick random value $[x]_1 \in \mathbb{Z}_{2^\ell}$;
  - P_0 and P_2 pick random value $[x]_2 \in \mathbb{Z}_{2^\ell}$;
  - P_0 calculates $x = [x]_1 + [x]_2$.
- $[\![x]\!] \leftarrow \Pi^p_{[\![\cdot]\!]}(x)$:

TABLE 2: The share structure of Aegis. (For $\|\cdot\|_i^{p,\lambda}$, the example in the table depicts the case of $\|\cdot\|_0^{p,\lambda}$)

	$[\![x]\!]^p$	$\|x\|_0^{p,\lambda}$	$[x]$	$\langle x\rangle$
P_0	x	$(x, \{\alpha_j\}_{j\in\mathbb{Z}_\lambda}, \{\gamma(x)_j\}_{j\in\mathbb{Z}_\lambda})$	x	$([r_x]_1, [r_x]_2 \in \mathbb{Z}_{2^\ell})$
P_1	$[\![x]\!]_1^p \in \mathbb{Z}_p$	$([\![x]\!]_1^p, \{[\![\alpha_j]\!]_1^p, [\![\gamma(x)_j]\!]_1^p\}_{j\in\mathbb{Z}_\lambda})$	$[x]_1 \in \mathbb{Z}_{2^\ell}$	$([r_x]_1, m_x = r_x + x)$
P_2	$[\![x]\!]_2^p \in \mathbb{Z}_p$	$([\![x]\!]_2^p, \{[\![\alpha_j]\!]_1^p, [\![\gamma(x)_j]\!]_2^p\}_{j\in\mathbb{Z}_\lambda})$	$[x]_2 \in \mathbb{Z}_{2^\ell}$	$([r_x]_2, m_x = r_x + x)$

- P_0 and P_1 pick random value $[\![x]\!]_1 \in \mathbb{Z}_p$;
- P_0 sends $[\![x]\!]_2^p = x - [\![x]\!]_1^p \pmod p$ to P_2.

- $[\![x]\!]^p \leftarrow \Pi^p_{[\![\cdot]\!]}$:
  - P_0 and P_1 pick random value $[\![x]\!]_1^p \in \mathbb{F}_p$;
  - P_0 and P_2 pick random value $[\![x]\!]_2^p \in \mathbb{F}_p$;
  - P_0 calculates $x = [\![x]\!]_1^p + [\![x]\!]_2^p$.
- $\langle x\rangle \leftarrow \Pi_{\langle\cdot\rangle}(x, P_i)$:
  - All parties perform $[r_x] \leftarrow \Pi_{[\cdot]}$ in the offline phase;
  - P_i send $m_x = x + r_x$ to P_1 and P_2.
- $\langle x\rangle \leftarrow \Pi^\ell_{\langle\cdot\rangle}$:
  - All parties perform $[r_x] \leftarrow \Pi_{[\cdot]}$ in the offline phase;
  - P_1 and P_2 pick random value m_x together.
- $\|x\| \leftarrow \Pi^{p,\lambda}_{\|\cdot\|}(x, P_i)$:
  - All parties invoke $[\![\alpha_j]\!]^p \leftarrow \Pi^p_{[\![\cdot]\!]}$ for $j \in \mathbb{Z}_\lambda$;
  - P_i calculates $\gamma(x)_j = x \cdot \alpha_j$ for $j \in \mathbb{Z}_\lambda$;
  - All parties invoke $[\![\gamma(x)_j]\!]^p \leftarrow \Pi^p_{[\![\cdot]\!]}(\gamma(x)_j)$ for $j \in \mathbb{Z}_\lambda$ and $[\![x]\!]^p \leftarrow \Pi^p_{[\![\cdot]\!]}(x)$.

$\Pi_{[\cdot]}$ and $\Pi_{\langle\cdot\rangle}$ also work for the share $[\cdot]^{\ell[x]}, \langle\cdot\rangle^{\ell[x]}$ over the polynomial ring $\mathbb{Z}_{2^\ell}[x]/f(x)$, which are denoted as $\Pi^{\ell[x]}_{[\cdot]}$, $\Pi^{\ell[x]}_{\langle\cdot\rangle}$.

Robustness of reconstruction. We note that the shared form $\langle\cdot\rangle$ has the reconstruction robustness property against a single malicious party. To be precise, for shared value $\langle x\rangle$, a single active adversary cannot deceive the honest parties into accepting an incorrect reconstruction result $x+e$ with a non-zero error e. This is because any two honest parties can collaboratively reconstruct the secret, and invalid shares will be detected by the honest parties. In addition, the shared form $\|\cdot\|_i^p$ also maintains the robustness when one of the P_{i-1}, P_{i+1} is malicious. Because P_i can assert the correctness of share through the MAC check. Formally, the robust reconstruction protocol Π_{Rec} is described as follows:

- $x \leftarrow \Pi_{\mathsf{Rec}}(\langle x\rangle)$:
  - P_0 sends $[r_x]_1$ to P_2 and $[r_x]_2$ to P_1;
  - P_1 sends m_x to P_0 and $[r_x]_1$ to P_2;
  - P_2 sends m_x to P_0 and $[r_x]_2$ to P_1;

  If the received messages from the other parties are inconsistent, P_i output abort. Otherwise P_i output $x = m_x - [r_x]_1 - [r_x]_2$.
- $x \leftarrow \Pi_{\mathsf{Rec}}(\langle x\rangle, P_i)$: All parties send their shares to P_i. If the received messages from the other parties are inconsistent, P_i output abort. Otherwise P_i output $x = m_x - [r_x]_1 - [r_x]_2$.
- $x \leftarrow \Pi^p_{\mathsf{Rec}}(\|x\|, P_i)$:
  - Each party P_k for $k \neq i$ sends its shares $[\![x]\!]_k^p, \{[\![\gamma(x)_j]\!]_k^p\}_{j\in\mathbb{Z}_\lambda}$ to P_i;
  - P_i reconstructs x and $\{\gamma(x)_j\}_{j\in\mathbb{Z}_\lambda}$, aborts if any $\gamma(x)_j \neq \alpha_j \cdot x$ for $j \in \mathbb{Z}_\lambda$.

For the share $\langle\cdot\rangle^{\ell[x]}$ in polynomial ring, $\Pi^{\ell[x]}_{\mathsf{Rec}}$ 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 $[r_z] \leftarrow \Pi_{[\cdot]}(r_z)$ for the output wire in the offline phase. To calculate m_z for P_1 and P_2 in the online phase, it can be written as

$$\begin{aligned} m_z = xy + r_z &= (m_x - r_x)(m_y - r_y) + r_z \\ &= m_x m_y - m_x r_y - m_y r_x + r_x r_y + r_z \ . \end{aligned}$$

$[\Gamma'] = m_x m_y - m_x[r_y] - m_y[r_x]$ can be calculated by P_1 and P_2 locally and $[\Gamma] = [r_x\cdot r_y] - [r_z]$ 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 $[m_z] = [\Gamma'] + [\Gamma]$.

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

$$m_y = F^d(m_{x_1} - r_{x_1},\ldots,m_{x_n} - r_{x_n}) + r_y \tag{1}$$

Let $\mathcal{I}_k$ be the k^{th} item of $F^d(x_1,\ldots,x_n) = \sum_{k=0}^m c_k \cdot \prod_{x_{s_j}\in\mathcal{I}_k} x_{s_j}$. After expanding Eq. 1, we let P_0 locally computes all the cross-items $\prod_{x_{s_j}\in\mathcal{I}_k} r_{x_{s_j}}$ and share them to the other parties in the offline phase. The offline phase requires ℓm bits communication depending on the number of cross-items, i.e. m. Let $\Pi^{\ell[x]}_{\mathsf{PolyEvl}}$ denote the polynomial evaluation protocol w.r.t. a polynomial ring $\mathbb{Z}_{2^\ell}[x]/f(x)$. Analogously, it costs $2\ell d$ of communication in the online phase and at most $\ell\cdot d\cdot m$ in the offline phase, for the degree d of $f(x)$.

Security up to additive attacks. As proven in [10], an replicated secret sharing protocol, such as Π_{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 Composibility (UC) framework [7], which follows the simulation-based security paradigm. The adversary $\mathcal{A}$ 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 Π is a secure realization of the functionality $\mathcal{F}$, if it satisfies that for every PPT adversary $\mathcal{A}$ attacking an execution of Π, there is another PPT adversary $\mathcal{S}$ (simulator) attacking the ideal process that uses $\mathcal{F}$ where the executions of Π with $\mathcal{A}$ and that of $\mathcal{F}$ with $\mathcal{S}$ makes no difference to any PPT environment $\mathcal{Z}$.

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

Functionality $\mathcal{F}_{\mathsf{3pc}}$

$\mathcal{F}_{\mathsf{3pc}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. Let f denote the functionality to be computed.

Input:

- Upon receiving from $(\mathsf{Input}, \mathsf{sid}, x_i)$ from $P_i \in \mathcal{P}$, record x_i and send $(\mathsf{Input}, \mathsf{sid}, P_i)$ to $\mathcal{S}$.

Execution:

- Upon receiving $(\mathsf{Compute}, \mathsf{sid})$ from $\mathcal{S}$, if all x_i are recorded compute $(y_0, y_1, y_2) = f(x_0, x_1, x_2)$.
- For $i \in [3]$, send (Output, y_i) to P_i via *private delayed channel*.

Figure 2: The ideal functionality $\mathcal{F}_{\mathsf{3pc}}$.

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 $\mathcal{F}_{\mathsf{sc}}$ for the protocol execution Π. 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 Π UC-secure realizes functionality $\mathcal{F}$ if for all PPT adversaries $\mathcal{A}$ there exists a PPT simulator $\mathcal{S}$ such that for all PPT environment $\mathcal{Z}$ it holds:

$$\mathsf{Real}_{\Pi,\mathcal{A},\mathcal{Z}}(1^\lambda) \approx \mathsf{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(1^\lambda)$$

4. Secure Sign Bit Extraction

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

Let $\mathcal{L}_1 := \{s_{|j}\}_{j \in \mathbb{Z}_\ell}$ be the list of the individual bits of the shared value s. One can transfer $\mathcal{L}_1$ into another list $\mathcal{L}_2 := \{t_{|j}\}_{j \in \mathbb{Z}_\ell}$ such that only one zero-element exists at

Protocol $\Pi_{\mathsf{SignBit}}(\langle x \rangle)$

P_1 and P_2 hold a common random seed $\eta \in \{0,1\}^\lambda$.
Input : $\langle \cdot \rangle$-shared value of x.
Output : $\langle \cdot \rangle$-shared value of $z = \mathsf{sign}(x)$.

Preprocessing:

- All parties perform $[r'], [r_z] \leftarrow \Pi_{[\cdot]}$;
- P_i, for $i \in \{1,2\}$ generates the same random value $\Delta \in \{0,1\}$ via PRF with seed η and reveals $[\Gamma] = \Delta + [r'] - 2\Delta \cdot [r'] + [r_z]$ to each other.
- P_0 does:
  1) calculate $\hat{r_x} = -r_x - \mathsf{sign}(-r_x) \cdot 2^{\ell-1} \in \mathbb{Z}_{2^\ell}$
  2) extract $2^{\ell-1} - 1 - \hat{r_x}$ as $\{r_{x,0}, \ldots, r_{x,\ell-1}\}$
  3) perform $\llbracket r_{x,j} \rrbracket^p \leftarrow \Pi^p_{\llbracket \cdot \rrbracket}(r_{x,j})$ for $j \in \mathbb{Z}^*_\ell$, taking the biggest prime of $p \in (\ell, 2^{\log \ell + 1}]$;

Online:

- P_i, for $i \in \{1,2\}$ does:
  1) set $\hat{m_x} = m_x - \mathsf{sign}(m_x) \cdot 2^{\ell-1}$ and bitexact it as $\{\hat{m}_{x|j} \in \{0,1\}\}_{j \in \mathbb{Z}_\ell}$ while $\sum_{j=0}^{\ell-1} 2^{\ell-1-j} \hat{m}_{x|j} = \hat{m_x}$;
  2) set $\hat{m}_{x|\ell} = 0$ and $\llbracket r_{x,\ell} \rrbracket = \llbracket 1 \rrbracket$;
  3) set $\llbracket m_j \rrbracket^p = \hat{m}_{x|j} + \llbracket r_{x,j} \rrbracket^p - 2\hat{m}_{x|j} \cdot \llbracket r_{x,j} \rrbracket^p$ for $j \in \mathbb{Z}^*_{\ell+1}$.
  4) pick same random values $\{w_j, w'_j \in \mathbb{Z}^*_p\}_{j \in \mathbb{Z}^*_{\ell+1}}$ via PRF with seed η;
  5) calculate $\llbracket m'_j \rrbracket^p = \sum_{k=1}^{j} \llbracket m_k \rrbracket^p - 2 \cdot \llbracket m_j \rrbracket^p + 1$ and $\llbracket u_j \rrbracket^p = w_j \cdot \llbracket m'_j \rrbracket^p + (\mathsf{sign}(m_x) \oplus \hat{m}_{x|j} \oplus \Delta)$ and $\llbracket u'_j \rrbracket^p = w'_j(w_j \cdot \llbracket m'_j \rrbracket^p + 1)$ for $j \in \mathbb{Z}^*_{\ell+1}$;
  6) pick a random permutation π via PRF with seed η and permute the list $\{\llbracket \hat{u_j} \rrbracket^p\}_{j \in \mathbb{Z}^*_{\ell+1}} = \pi(\{\llbracket u_j \rrbracket^p\}_{j \in \mathbb{Z}^*_{\ell+1}})$ and $\{\llbracket \hat{u'_j} \rrbracket^p\}_{j \in \mathbb{Z}^*_{\ell+1}} = \pi(\{\llbracket u'_j \rrbracket^p\}_{j \in \mathbb{Z}^*_{\ell+1}})$;
  7) reveal $\{\llbracket \hat{u_j} \rrbracket^p\}_{j \in \mathbb{Z}^*_{\ell+1}}$ and $\{\llbracket \hat{u'_j} \rrbracket^p\}_{j \in \mathbb{Z}^*_{\ell+1}}$ to P_0;
- P_0 sends $m' = \mathsf{sign}(-r_x) - r'$ if $\exists \hat{u_j} = 0 \wedge \hat{u'_j} \neq 0$ for $j \in \mathbb{Z}^*_{\ell+1}$ else $m' = (1 \oplus \mathsf{sign}(-r_x)) - r'$ to P_i, for $i \in \{1,2\}$;
- P_i, for $i \in \{1,2\}$ sets $m_z = m' - 2\Delta \cdot m' + \Gamma$;
- All parties output $\langle z \rangle = ([r_z], m_z)$.

Figure 3: The Sign Bit Extraction Protocol.

the position corresponding to the first non-zero bit of $\mathcal{L}_1$. Namely, $t_{|j} = \sum_{k=0}^{j} s_{|k} - 2 \cdot s_{|j} + 1 \bmod p$ for $j \in \mathbb{Z}_\ell$, where $p \geq \ell$. We denote this transform as ϕ. The intuition of our sign bit extraction is as follows.

Let $m_x := \mathsf{sign}(m_x) || \hat{m_x}$ and $-r_x := \mathsf{sign}(-r_x) || \hat{r_x}$. Instead of extracting the sign bit, we evaluate $\mathsf{sign}(x) := (\hat{m_x} + \hat{r_x} \geq 2^{\ell-1}) \oplus \mathsf{sign}(-r_x) \oplus \mathsf{sign}(m_x)$. To calcuate $\hat{m_x} + \hat{r_x} \geq 2^{\ell-1}$, we evaluate $\hat{m_x} \overset{?}{>} 2^{\ell-1} - 1 - \hat{r_x}$ (It works due to $2^{\ell-1} - 1 \geq \hat{r_x}$). If we consider $\hat{m_x}$ and $2^{\ell-1} - 1 - \hat{r_x}$ 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 $\hat{m_x}$ and $2^{\ell-1} - 1 - \hat{r_x}$, which indicates the position where the two values diverge. We have $\hat{m}_{x|\mathsf{ind}} = \hat{m_x} \overset{?}{>} 2^{\ell-1} - 1 - \hat{r_x}$.

Following this intuition, we apply $\{m'_j\}_{j \in \mathbb{Z}_\ell} = \phi(\{m_{|j}\}_{j \in \mathbb{Z}_\ell})$ to identify the first nonzero bit. As ϕ operates over field $\mathbb{Z}_p$, we first transfer $\{m_{|j}\}_{j \in \mathbb{Z}_\ell}$ to shares over

Protocol $\Pi_{\mathsf{Trans}}(\{\langle x_i\rangle, \langle y_i\rangle, \langle z_i\rangle\}_{i\in\mathbb{Z}_N})$

Input : N triples of $\langle\cdot\rangle$-shared multiplication.
Output : One triple of N-dimension $\langle\cdot\rangle^{\ell[x]}$-shared inner product.

Preprocessing:

- All parties invoke $\langle r\rangle^{\ell[x]} \leftarrow \Pi^{\ell[x]}_{\langle\cdot\rangle}$ locally;

Online:

- All parties reconstruct r with Π_{Rec} and calculate r^i for all $i \in \mathbb{Z}_N$;
- All parties transfer $\langle\cdot\rangle$ to $\langle\cdot\rangle^{\ell[x]}$ locally by setting the constant term of $\langle\cdot\rangle^{\ell[x]}$ to $\langle\cdot\rangle$;
- All parties set $\langle z\rangle^{\ell[x]} := \sum_{i=0}^{N-1} r^i \cdot \langle z_i\rangle^{\ell[x]}$, and $\langle x'_i\rangle^{\ell[x]} := r^i \cdot \langle x_i\rangle^{\ell[x]}$ for all $i \in \mathbb{Z}_N$;
- All parties output $\{\langle x'_i\rangle^{\ell[x]}, \langle y_i\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_N}; \langle z\rangle^{\ell[x]}$.

Figure 4: Compression of Multiplication Triples.

$\mathbb{Z}_p$. To further compute $\hat{m}_{x|\mathsf{ind}}$, we add vector $\{m'_j\}_{j\in\mathbb{Z}_\ell}$ to $\{\hat{m}_{x|j}\}_{j\in\mathbb{Z}_\ell}$, denote the resulting new vector as $\{u_j\}_{j\in\mathbb{Z}_\ell}$. There are two possible situations: (i) $\{u_j\}_{j\in\mathbb{Z}_\ell}$ does not contain 0, means that $\hat{m}_{x|\mathsf{ind}} = 1$; (ii) $\{u_j\}_{j\in\mathbb{Z}_\ell}$ contains 0, means that the 0 is derived from $\hat{m}_{x|\mathsf{ind}} = 0$ or $\hat{m}_{x|\mathsf{ind}} = 1 \wedge m'_j = -1$. If we exclude the situation of $m'_j = -1$, we can determine the sign bit by checking whether $\{u_j\}_{j\in\mathbb{Z}_\ell}$ contains 0. Our protocol let P_1 and P_2 generate $\{u_j\}_{j\in\mathbb{Z}_\ell}$, and let P_0 check whether $\{u_j\}_{j\in\mathbb{Z}_\ell}$ contains 0. To protect the privacy, we let P_1 and P_2 locally permute the $\{u_j\}_{j\in\mathbb{Z}_\ell}$ list and mask $\hat{m}_{x|\mathsf{ind}}$ with a random bit Δ. Considering $\mathsf{sign}(x) := (\hat{m}_x + \hat{r}_x \geq 2^{\ell-1}) \oplus \mathsf{sign}(-r_x) \oplus \mathsf{sign}(m_x)$, we make $\hat{m}_{x|\mathsf{ind}}$ further XOR $\mathsf{sign}(m_x)$. Finally, we utilize list $\{u'_j\}_{j\in\mathbb{Z}_\ell}$ to exclude the situation of $m'_j = -1$. Formally, our protocol is described in Fig. 3. The procedures are as follows:

- P_1 and P_2 set $[\![m_j]\!]^p$, where m_j represents the j-th bit of $\hat{m}_x \oplus (2^{\ell-1} - 1 - \hat{r}_x)$. The transformation can be locally performed as outlined in Fig. 3 (Steps 1-3). Moreover, we set $\hat{m}_{x|\ell} = 0$ and $[\![r_{x,\ell}]\!] = [\![1]\!]$ to ensure that protocol output equals to 0 when $\hat{m}_x + \hat{r}_x = 2^{\ell-1} - 1$.
- P_1, P_2 transfer $[\![m_j]\!]^p$ to $[\![m'_j]\!]^p$ via the aforementioned transformation ϕ and calculate $[\![u_j]\!]^p = w_j \cdot [\![m'_j]\!]^p + (\mathsf{sign}(m_x) \oplus \hat{m}_{x|j} \oplus \Delta)$ with the random list w_j and the masked value $\mathsf{sign}(m_x) \oplus \hat{m}_{x|j} \oplus \Delta$.
- P_1, P_2 open $\{u_j\}_{j\in|\ell|}$ to P_0, and P_0 can draw conclusions based on observations of $\{u_j\}_{j\in|\ell|}$.
  - If there exist j that $u_j = 0$, then either $\mathsf{sign}(m_x) \oplus \hat{m}_{x|j} \oplus \Delta = 0$ or $(\mathsf{sign}(m_x) \oplus \hat{m}_{x|j} \oplus \Delta = 1) \wedge (w_j \cdot [\![m'_j]\!]^p = p-1)$.
  - If there $\nexists j$ such that $u_j = 0$, then $\mathsf{sign}(m_x) \oplus \hat{m}_{x|j} \oplus \Delta = 1$.
- Next, we exclude the cases where $w_j \cdot [\![m'_j]\!]^p = p-1$ as follows. P_1 and P_2 calculate $[\![u'_j]\!]^p = w'_j \cdot (w_j \cdot [\![m'_j]\!]^p + 1)$ and open u'_j to P_0. (Note that $u'_j = 0$ if $w_j \cdot [\![m'_j]\!]^p = p-1$.) P_0 then can set $\mathsf{sign}(m_x) \oplus \hat{m}_{x|j} \oplus \Delta$ as: if there exist j that $u_j = 0 \wedge u'_j \neq 0$, then $\mathsf{sign}(m_x) \oplus \hat{m}_{x|j} \oplus \Delta = 0$, otherwise $\mathsf{sign}(m_x) \oplus \hat{m}_{x|j} \oplus \Delta = 1$.
- Now, P_0 holds $\mathsf{sign}(m_x) \oplus \hat{m}_{x|j} \oplus \Delta$. P_1 and P_2 hold $\Delta \oplus \mathsf{sign}(-r_x)$. We further introduce $[r']$ where r' is known to P_0 to transfer $\{\mathsf{sign}(m_x) \oplus \hat{m}_{x|j} \oplus \Delta, \Delta \oplus \mathsf{sign}(-r_x)\}$ to $\langle z\rangle$. Specifically, we first let P_1 and P_2 reveal $[\Gamma] = \Delta + [r'] - 2\Delta \cdot [r'] + [r_z]$ to each other in the offline phase. Thanks to random r', P_1 and P_2 learn nothing about r_z. After that, P_0 sends $m' = \mathsf{sign}(m_x) \oplus \hat{m}_{x|j} \oplus \Delta \oplus \mathsf{sign}(-r_x) - r'$ to both P_1 and P_2. P_1 and P_2 then locally calculate $m_z = m' - 2\Delta m' + \Gamma$. We can verify that $m_z - r_z = (\hat{m}_x + \hat{r}_x \geq 2^{\ell-1}) \oplus \mathsf{sign}(-r_x) \oplus \mathsf{sign}(m_x) = \mathsf{sign}(x)$.

Protocol $\Pi_{\mathsf{Reduce}}(\{\langle x_i\rangle^{\ell[x]}, \langle y_i\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_N}, \langle z\rangle^{\ell[x]})$

Input : N-dimension $\langle\cdot\rangle^{\ell[x]}$-shared inner product.
Output : $N/2$-dimension $\langle\cdot\rangle^{\ell[x]}$-shared inner product.

Execution:

- For $i \in \mathbb{Z}_{N/2}$, all parties set
  - $\langle f_i(0)\rangle^{\ell[x]} = \langle x_{2\cdot i}\rangle^{\ell[x]}$;$\langle f_i(1)\rangle^{\ell[x]} = \langle x_{2\cdot i+1}\rangle$; $\langle f_i(2)\rangle^{\ell[x]} = 2 \cdot \langle f_i(1)\rangle^{\ell[x]} - \langle f_i(0)\rangle^{\ell[x]}$;
  - $\langle g_i(0)\rangle^{\ell[x]} = \langle x_{2\cdot i}\rangle^{\ell[x]}$;$\langle g_i(1)\rangle^{\ell[x]} = \langle x_{2\cdot i+1}\rangle^{\ell[x]}$; $\langle g_i(2)\rangle^{\ell[x]} = 2 \cdot \langle g_i(1)\rangle^{\ell[x]} - \langle g_i(0)\rangle^{\ell[x]}$;
  - $\langle h(0)\rangle^{\ell[x]} = \sum \langle f_i(0)\rangle^{\ell[x]} \cdot \langle g_i(0)\rangle^{\ell[x]}$;$\langle h(1)\rangle^{\ell[x]} = \langle z\rangle^{\ell[x]} - \langle h(0)\rangle^{\ell[x]}$; $\langle h(2)\rangle^{\ell[x]} = \sum \langle f_i(2)\rangle^{\ell[x]} \cdot \langle g_i(2)\rangle^{\ell[x]}$;
- All parties invoke $\langle\zeta\rangle^{\ell[x]} \leftarrow \Pi^{\ell[x]}_{\langle\cdot\rangle}$ and reveal $\langle 2\cdot\zeta\rangle^{\ell[x]}$;
- All parties calculate
  - $\langle h(\zeta)\rangle^{\ell[x]} = \sum_{i=0}^{2}((\Pi_{j=1,j\neq i}^{2} \frac{\zeta-j}{i-j}) \cdot \langle h(i)\rangle^{\ell[x]})$;
  - $\langle f_i(\zeta)\rangle^{\ell[x]} = \zeta \cdot \langle f_i(1)\rangle^{\ell[x]} - (\zeta - 1)\langle f_i(0)\rangle^{\ell[x]}$;
  - $\langle g_i(\zeta)\rangle^{\ell[x]} = \zeta \cdot \langle g_i(1)\rangle^{\ell[x]} - (\zeta - 1)\langle g_i(0)\rangle^{\ell[x]}$;
- All parties output $\{\langle f_i(\zeta)\rangle^{\ell[x]}, \langle g_i(\zeta)\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_{N/2}}; \langle h(\zeta)\rangle^{\ell[x]}$.

Figure 5: The Inner Product Dimension Reduction Protocol

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

Security. We analyze the security of our sign-bit extraction protocol in the UC framework. The functionality $\mathcal{F}_{\mathsf{SignBit}}$ for sign bit extraction is defined as follows. As an instantiation of $\mathcal{F}_{\mathsf{3pc}}$ depicted in Fig. 2, $\mathcal{F}_{\mathsf{SignBit}}$ receives (Input,sid,r_x) from P_0, (Input,sid,m_x) from P_1, (Input,sid,m_x) from P_2. It calculates $z = \mathsf{sign}(m_x - r_x)$. If P_0 is corrupted, $\mathcal{F}_{\mathsf{SignBit}}$ obtains $[r_x]_1$ and $[r_x]_2$ from $\mathcal{S}$. If P_i for $i = 1$ or $i = 2$ is corrupted, $\mathcal{F}_{\mathsf{SignBit}}$ obtains $[r_x]_i$ from $\mathcal{S}$ and picks random value $[r_z]_{3-i} \in \mathbb{Z}_{2^\ell}$; $\mathcal{F}_{\mathsf{SignBit}}$ sets $m_z = z + [r_z]_1 + [r_z]_2$ and sends $(\mathsf{Output}, [r_z]_1, [r_z]_2)$ to P_0, $(\mathsf{Output}, [r_z]_1, m_z)$ to P_1, $(\mathsf{Output}, [r_z]_2, m_z)$ to P_2.

Theorem 1. *Let $\mathsf{PRF}^{(\mathbb{Z}_p)^p}$,$\mathsf{PRF}^{\mathbb{Z}_p}$ and $\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}$ be the secure pseudo-random functions. The protocol Π_{SignBit} as depicted in Fig. 3 UC realizes $\mathcal{F}_{\mathsf{SignBit}}$ against semi-honest PPT adversaries who can statically corrupt up to one party.*

Protocol $\Pi_{\textsf{InnerVerify}}(\{\langle x_i\rangle^{\ell[x]}, \langle y_i\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_N}, \langle z\rangle^{\ell[x]})$

Input : A N-dimension $\langle\cdot\rangle^{\ell[x]}$-shared inner product pair.
Output : $z \stackrel{?}{=} \sum_{i=1}^{N} x_i \cdot y_i$.
Execution:

- All parties invoke $\langle\alpha\rangle^{\ell[x]} \leftarrow \Pi_{\langle\cdot\rangle}^{\ell[x]}$;
- All parties calculate $\langle\Delta\rangle^{\ell[x]} = \langle\alpha\rangle^{\ell[x]} \cdot (\sum_{i=1}^{N} \langle x_i\rangle^{\ell[x]} \cdot \langle y_i\rangle^{\ell[x]} - \langle z\rangle^{\ell[x]})$ with $\Pi_{\textsf{PolyEvl}}^{\ell[x]}$;
- All parties call $\Delta = \Pi_{\textsf{Rec}}^{\ell[x]}(\langle\Delta\rangle^{\ell[x]})$;
- All parties output 1 if $\Delta = 0$, otherwise 0.

Figure 6: The Inner Product Verification Protocol

Proof. See Appendix A.1. □

Protocol $\Pi_{\textsf{MultVerify}}^{R}(\{\langle x_i\rangle, \langle y_i\rangle, \langle z_i\rangle\}_{i\in\mathbb{Z}_N})$

Input : N pairs of $\langle\cdot\rangle$-shared multiplication.
Output : $z_i \stackrel{?}{=} x_i \cdot y_i$ for all $i \in \mathbb{Z}_N$.
Execution:

- All parties invoke $\Pi_{\textsf{Trans}}(\{\langle x_i\rangle, \langle y_i\rangle; \langle z_i\rangle\}_{i\in\mathbb{Z}_N})$ to get $\{\langle x_i\rangle^{\ell[x]}, \langle y_i\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_N}; \langle z\rangle^{\ell[x]}$;
- For $k = 1, \ldots, R$, all parties perform:
  - $\{\{\langle x_i\rangle^{\ell[x]}, \langle y_i\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_{N/2^k}}; \langle z\rangle^{\ell[x]}\} \leftarrow \Pi_{\textsf{Reduce}}(\{\langle x_i\rangle^{\ell[x]}, \langle y_i\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_{N/2^{k-1}}}; \langle z\rangle^{\ell[x]})$;
- All parties invoke $b = \Pi_{\textsf{InnerVerify}}(\{\langle x_i\rangle^{\ell[x]}, \langle y_i\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_{N/2^R}}; \langle z\rangle^{\ell[x]})$;
- All parties output b.

Figure 7: The Batch Multiplication Verification Protocol

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 $\{\langle x_i\rangle, \langle y_i\rangle, \langle z_i\rangle\}_{i\in\mathbb{Z}_N}$ to the task of verifying the inner product $\Delta = \sum_{i=0}^{N} \langle r^i \cdot x_i\rangle \cdot \langle y_i\rangle - \langle r^i \cdot z_i\rangle$ 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 $\langle z_i\rangle$, allowing her to cancel out the error when computing Δ 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 $e = 2^{\ell-1}$ 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 $\Delta = \langle\alpha\rangle \cdot (\sum_{i=0}^{N} \langle r^i \cdot x_i\rangle \cdot \langle y_i\rangle - \langle r^i \cdot z_i\rangle)$ (using $\Pi_{\textsf{PolyEvl}}$) with random share $\langle\alpha\rangle$. Since $\Pi_{\textsf{PolyEvl}}$ is secure up to additive attack [10], the adversary can only introduce an input-independent additive error e' of Δ. Therefore, the adversary has to guess $e' = e \cdot \alpha$ to make $\Delta = 0$ with the probability $2^{-\ell}$. To resolve the latter issue, we perform Δ over the extension ring $\mathbb{Z}_{2^\ell}[x]/f(x)$, where $f(x)$ is a degree-d irreducible polynomial over $\mathbb{Z}_2$ [3]. (This can be done by putting the original share over $\mathbb{Z}_{2^\ell}$ 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 $\frac{2^{(\ell-1)d}N+1}{2^{\ell d}} \approx \frac{N}{2^d}$ 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 $\Theta(\log N \cdot d)$. Following this idea, we construct our batch multiplication verification protocol as follows.

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

Lemma 1. *Suppose protocol $\Pi_{\textsf{Trans}}$ depicted in Fig. 4 take input as $\{\langle x_i\rangle, \langle y_i\rangle, \langle z_i\rangle\}_{i\in\mathbb{Z}_N}$, and it outputs $\{\langle x'_i\rangle^{\ell[x]}, \langle y_i\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_N}; \langle z\rangle^{\ell[x]}$. 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. $\mathbb{Z}_{2^\ell}[x]/f(x)$:*

- $z = \sum_{i=0}^{N} x'_i \cdot y_i$
- $\exists i \in \mathbb{Z}_N$ *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 $\Pi_{\textsf{Reduce}}$ takes input as a shared triple $(\{\langle x_i\rangle^{\ell[x]}, \langle y_i\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_N}, \langle z\rangle^{\ell[x]})$ and outputs the shared triple $(\{\langle x'_i\rangle^{\ell[x]}, \langle y'_i\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_{N/2}}, \langle z'\rangle^{\ell[x]})$. $\Pi_{\textsf{Reduce}}$ ensures that $\sum_{i=0}^{N} x_i \cdot y_i = z$ if and only if $\sum_{i=0}^{N/2} x'_i \cdot y'_i = z'$ except for a negligible probability. At a high level, for the inner product input $\{x_i\}_{i\in\mathbb{Z}_N}$ and $\{y_i\}_{i\in\mathbb{Z}_N}$, we can utilize x_{2i} and x_{2i-1} to interpolate $N/2$ linear functions $\{f_i(\cdot)\}_{i\in\mathbb{Z}_{N/2}}$ at the point 0 and 1, and similarly interpolate $\{g_i(\cdot)\}_{i\in\mathbb{Z}_{N/2}}$ 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 $h(\cdot) = \sum_{i=0}^{N/2} f_i(\cdot) \cdot g_i(\cdot)$. The above equation can be written as

Protocol $\Pi^{\lambda}_{\mathsf{Pos}}(\langle x\rangle, P_i)$

Input : $\langle\cdot\rangle$-shared value of x.
Output : 1 if the sign bit of x is 0, 0 otherwise.
Execution:

- Parse $\langle x\rangle := \{m_x, [r_x]_1, [r_x]_2\}$ as $\{x_0, -x_1, -x_2\}$ where P_k hold x_k and $x_0 + x_1 + x_2 = x$;
- The verifier P_i calculates $r = x_{i-1} + x_{i+1} - \mathsf{sign}(x_{i-1} + x_{i+1}) \cdot 2^{\ell-1}$. Then P_i chops $2^{\ell-1} - 1 - r$ as $\{r_0, \ldots, r_{\ell-1}\}$, and sets $r_\ell = 1$;
- All parties performs $\|r_j\| \leftarrow \Pi^p_{\|\cdot\|}(r_j, P_i)$ for $j \in \mathbb{Z}^*_{\ell+1}$, taking the biggest prime of $p \in (\ell, 2^{\log \ell+1}]$;
- P_{i-1} and P_{i+1} do:
  1) pick random list $w_j, w'_j \in \mathbb{Z}_p$ for $j \in \mathbb{Z}_{\ell+1}$;
  2) generate a random shift π together;
  3) set $m_\sigma = x_i - \mathsf{sign}(x_i) \cdot 2^{\ell-1}$, bitexact it as $\{m_{\sigma|j}\}_{j\in\mathbb{Z}_\ell}$ and set $m_{\sigma|\ell} = 0$;
  4) calculate $\|m_j\| = m_{\sigma|j} + \|r_j\| - 2m_{\sigma|j} \cdot \|r_j\|$ and $\|m'_j\| = \sum_{k=0}^{j} \|m_k\| - 2 \cdot \|m_j\| + 1$ for $j \in \mathbb{Z}^*_{\ell+1}$;
  5) calculate $\|u_j\| = \pi(w_j \cdot \|m'_j\| + m_{\sigma|j} \oplus \mathsf{sign}(x_i))$ and $\|u'_j\| = \pi(w'_j(w_j \cdot \|m'_j\| - (p-1)))$ for $j \in \mathbb{Z}^*_{\ell+1}$;
- All parties invoke $u_j = \Pi^p_{\mathsf{Rec}}(\|u_j\|, P_i)$, $u'_j = \Pi^p_{\mathsf{Rec}}(\|u'_j\|, P_i)$. Consequently, P_i holds u_j,u'_j for $j \in \mathbb{Z}^*_{\ell+1}$;
- P_i output $1 \oplus \mathsf{sign}(x_{i-1} + x_{i+1})$ if $\exists u_j = 0 \wedge u'_j \neq 0$ for $j \in \mathbb{Z}_{\ell+1}$.

Figure 8: Positive Verification Protocol Verified by P_i.

$h(1) = z - h(0)$. Π_{Reduce} evaluates $h(0) = \sum_{i=0}^{N/2} f_i(0) \cdot g_i(0)$ and $h(2) = \sum_{i=0}^{N/2} f_i(2) \cdot g_i(2)$; in addition, $h(1) = z - h(0)$. Subsequently, Π_{Reduce} utilizes $h(0)$, $h(1)$ and $h(2)$ to interpolate the resulting polynomial $h(x)$. Finally, we let all parties select a random point ζ, and output the new shared triple $(\{\langle f_i(\zeta)\rangle^{\ell[x]}, \langle g_i(\zeta)\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_{N/2}}, \langle h(\zeta)\rangle^{\ell[x]})$ which inheres the inner product relationship if and only if $z = \sum_{i=1}^{N/2} f_i(0) \cdot g_i(0) + f_i(1) \cdot g_i(1)$.

Note that points 0, 1, 2 refer to the ring elements with free coefficient of 0, 1, and 2 in $\mathbb{Z}_{2^\ell}[x]/f(x)$. It is easy to see that Π_{Reduce} requires one round communication of $8\ell \cdot d$ bits in the online phase and one round communication of $\ell \cdot d$ bits in the offline phase. We perform R times Π_{Reduce} to reduce the inner product to dimension $N/2^R$, and the resulting vectors are verified as $\sum_{i=0}^{N/2^R} \langle f_i(\zeta)\rangle^{\ell[x]} \cdot \langle g_i(\zeta)\rangle^{\ell[x]} = \langle h(\zeta)\rangle^{\ell[x]}$. We prove the soundness error of the Π_{Reduce} is $\frac{1}{2^{d-1}}$ in Lemma 2.

Lemma 2. *Suppose protocol Π_{Reduce} depicted in Fig. 5 take input as $(\{\langle x_i\rangle^{\ell[x]}, \langle y_i\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_N}, \langle z\rangle^{\ell[x]})$, and it outputs $(\{\langle x'_i\rangle^{\ell[x]}, \langle y'_i\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_{N/2}}, \langle z'\rangle^{\ell[x]})$. The probability that the following two conditions hold is at most $\frac{1}{2^{d-1}}$, where d is the degree of $f(x)$ w.r.t. $\mathbb{Z}_{2^\ell}[x]/f(x)$:*

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

Proof. See Appendix A.3. □

Protocol $\Pi_{\mathsf{VSignBit}}(\langle x\rangle)$

Input : $\langle\cdot\rangle$-shared value.
Output : $\langle\cdot\rangle$-shared value of $z = \mathsf{sign}(x)$.
Execution:

- All parties perform $\langle z\rangle \leftarrow \Pi_{\mathsf{SignBit}}(\langle x\rangle)$;

Postprocessing:
For N pairs evaluation result $\{\langle x_i\rangle, \langle z_i\rangle\}_{i\in\mathbb{Z}_N}$, all parties do:

- Calculate $\langle x'_i\rangle = \langle x_i\rangle - 2^\ell \langle z_i\rangle$
- Perform $\Pi^R_{\mathsf{MultVerify}}(\{\langle z_j\rangle, \langle z_j\rangle; \langle z_j\rangle\}_{j\in\mathbb{Z}_N})$;
- Call $\Pi_{\mathsf{Pos}}(\{\langle x'_i\rangle\}_{i\in\mathbb{Z}_N}, P_0)$ and $\Pi_{\mathsf{Pos}}(\{\langle x'_i\rangle\}_{i\in\mathbb{Z}_N}, P_1)$ simultaneously;
- All parties output 1 if both P_0, P_1 output 1.

Figure 9: The Malicious Sign Bit Extraction Protocol.

Inner product verification. Our inner product verification $\Pi_{\mathsf{InnerVerify}}$ (Fig. 6) verifies the inner product relationship of shared values over polynomial ring $\mathbb{Z}_{2^\ell}[x]/f(x)$. For verification of $\sum_{i=0}^{N} \langle x_i\rangle^{\ell[x]} \cdot \langle y_i\rangle^{\ell[x]} = \langle z\rangle^{\ell[x]}$, $\Pi_{\mathsf{InnerVerify}}$ turns to verify $\langle \alpha\rangle^{\ell[x]} \cdot (\sum_{i=0}^{N/2^R} \cdot \langle x_i\rangle^{\ell[x]} \cdot \langle y_i\rangle^{\ell[x]} - \langle z_i\rangle^{\ell[x]})$ equal to zero. We prove soundness error of the $\Pi_{\mathsf{InnerVerify}}$ is $\frac{1}{2^d}$ in Lemma 3.

Lemma 3. *Let $(\{\langle x_i\rangle^{\ell[x]}, \langle y_i\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_N}, \langle z\rangle^{\ell[x]})$ be the input of protocol $\Pi_{\mathsf{InnerVerify}}$ depicted in Fig. 6. The probability that $\Pi_{\mathsf{InnerVerify}}$ 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. $\mathbb{Z}_{2^\ell}[x]/f(x)$.*

Proof. See Appendix A.4. □

Our batch multiplication verification protocol $\Pi_{\mathsf{MultVerify}}$ in Fig. 7 integrates the above three subroutines, which requires one round communication of $(R + 3N/2^R + 1)\ell \cdot d$ bits in the offline phase and $R+2$-round communication of $(10R + 8)\ell \cdot d$ bits in the online phase for N multiplication triples. We prove soundness error of $\Pi_{\mathsf{MultVerify}}$ is $\frac{N}{2^{d-R-2}}$ in Thm. 2.

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

Proof. See Appendix A.5. □

5.2. Sign Bit Extraction Verification Protocol

We upgrade the sign bit extraction Π_{SignBit} to the malicious setting throughout the verification protocol Π_{VSignBit}. For a sign bit extraction pair $\{\langle x\rangle, \langle z\rangle\}$, the malicious adversary can introduce arbitrary errors to make $\mathsf{sign}(x) \neq b$. As shown in Fig. 9, we design the verification protocol Π_{VSignBit} 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) $x - 2^\ell \cdot z$ is positive. The former check can be realized by employing a maliciously secure multiplication protocol to confirm that

Protocol $\Pi_{\mathsf{Mult}}(\langle x\rangle, \langle y\rangle)$

Input : $\langle\cdot\rangle$-shared value x, y.
Output : $\langle\cdot\rangle$-shared value z where $z = x \cdot y$.

Preprocessing:

- All parties prepare $[r_z] \leftarrow \Pi_{[\cdot]}$ locally;
- P_0 calculates $\Gamma = r_x \cdot r_y + r_z$ and shares it with $\Pi_{[\cdot]}(\Gamma)$;

Online:

- P_j for $j \in \{1, 2\}$ calculates $[m_z]_j = (j-1)m_x \cdot m_y - m_x[r_y]_j - m_{y_i}[r_x]_j + [\Gamma]$ and mutually exchange their shares to reconstruct m_z.

Postprocessing:

- For all multiple gate wire value $\{\langle x_i\rangle, \langle y_i\rangle, \langle z_i\rangle\}_{i\in\mathbb{Z}_N}$, all parties call $\Pi_{\mathsf{MultVerify}}(\{\langle x_i\rangle, \langle y_i\rangle; \langle z_i\rangle\}_{i\in\mathbb{Z}_N}, R)$ to verify correctness.

Figure 10: The Multiplication Protocol

its square matches itself, i.e., $z \cdot z = z$ on the ring $\mathbb{Z}_{2^\ell}$, as $z^2 - z = 0$ only has the roots of 0 and 1 over ring $\mathbb{Z}_{2^\ell}$. For this check, we directly utilize the aforementioned protocol $\Pi_{\mathsf{MultVerify}}(\langle z\rangle, \langle z\rangle, \langle z\rangle)$.

For the latter check, we first design the positive assertion protocol Π_{Pos} which nominates a verifier P_i to verify the positive of a shared value. Π_{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 Π_{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 Π_{Pos}**.** As depicted in Fig. 8, the positive assertion protocol Π_{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 $x \overset{?}{<} 2^\ell$. Specifically, we introduce the IT-secure MAC to detect malicious behavior of P_{i-1} and P_{i+1}. We observe that the chopped shared bit $[\![r_{x,j}]\!]$ in Π_{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_{i-1} and P_{i+1}. Later, when P_{i-1} and P_{i+1} send back the opened vector $\{\|u_j\|\}_{j\in\mathbb{Z}_\ell}$ and $\{\|u'_j\|\}_{j\in\mathbb{Z}_\ell}$, P_i can check the correctness of them by the corresponding MAC.

To support dual execution of Π_{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 $\{x_0 = m_x, x_1 = -[r_x]_1, x_2 = -[r_x]_2\}$. Following that all parties perform same operation in Π_{SignBit} which replace $[\![r_{x,j}]\!]$ with $\|r_{x,j}\|$ to generate the the vector $\{\|u_j\|\}_{j\in\mathbb{Z}_\ell}$ and $\{\|u'_j\|\}_{j\in\mathbb{Z}_\ell}$. Since $\mathsf{sign}(x) = 0$ 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 $\exists u_j = 0 \wedge u'_j \neq 0$. The soundness error of $\Pi^{p,\lambda}_{\mathsf{Pos}}$ is $\frac{1}{2^{\lambda\log\ell+\lambda+\log\ell}}$.

Protocol $\Pi^R_{\mathsf{BIVerify}}(\{\{\langle x_i^{(j)}\rangle, \langle y_i^{(j)}\rangle\}_{i\in\mathbb{Z}_{n_j}}, \langle z^{(j)}\rangle\}_{j\in\mathbb{Z}_N})$

Input : N pairs of inner product.
Output : Output if $z^{(j)} = \sum_{i=1}^n x_i^{(j)} \cdot y_i^{(j)}$ held for all $j \in \mathbb{Z}_N$.

Execution:

- All parties transfer all shares $\langle\cdot\rangle$ to $\langle\cdot\rangle^{\ell[x]}$ locally;
- All parties invoke $\langle r\rangle^{\ell[x]} \leftarrow \Pi^{\ell[x]}_{\langle\cdot\rangle}$ an call Π_{Rec} to reconstruct $r \in \mathbb{Z}_{2^\ell}[x]$;
- All parties set $\langle z\rangle^{\ell[x]} := \sum r^j \cdot \langle z^{(j)}\rangle^{\ell[x]}$ and $\langle x_i^{(j)}\rangle^{\ell[x]} := r^j \cdot \langle x_i^{(j)}\rangle^{\ell[x]}$ for each $i \in \mathbb{Z}_{n_j}, j \in \mathbb{Z}_N$;
- All parties consolidate the original pairs into a single pair $\{\langle x_i\rangle^{\ell[x]}, \langle y_i\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_\mathcal{N}}; \langle z\rangle^{\ell[x]}$ where $\mathcal{N} = \sum_{j=0}^{N-1} n_j$;
- For $k = 1, \ldots, R$, all parties do:
  - $\{\langle x_i\rangle^{\ell[x]}, \langle y_i\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_{\mathcal{N}/2^k}}, \langle z\rangle^{\ell[x]} \leftarrow \Pi_{\mathsf{Reduce}}(\{\langle x_i\rangle^{\ell[x]}, \langle y_i\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_{\mathcal{N}/2^{k-1}}}, \langle z\rangle^{\ell[x]})$;
- All parties call $b = \Pi_{\mathsf{InnerVerify}}(\{\langle x_i\rangle^{\ell[x]}, \langle y_i\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_{\mathcal{N}/2^R}}, \langle z\rangle^{\ell[x]})$;
- All parties output b.

Figure 11: The Batch Inner Product Verification Protocol

Theorem 3. *Let $\langle x\rangle^\ell$ be the input of the protocol $\Pi^{p,\lambda}_{\mathsf{Pos}}$ depicted Fig. 8. The probability that $\Pi^{p,\lambda}_{\mathsf{Pos}}$ outputs 1 and* $\mathsf{sign}(x) = 1$ *is at most* $\frac{1}{2^{\lambda\log\ell+\lambda+\log\ell}}$.

Proof. See Appendix A.6. $\square$

Our Sign bit extraction protocol Π_{VSignBit} requires amortized 2-round communication of $10\lambda\ell(\log\ell+1)+14\ell\log\ell+16\ell$ bits, where λ is MAC key number of $\|\cdot\|$.

6. The Aegis PPML Plaform

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. Π_{Mult} ensures the correctness of multiplication by invoking batch verification protocol $\Pi_{\mathsf{MultVerify}}$ in the post-processing phase. When handling a substantial volume of data, our protocol exhibits an amortized communication of ℓ bits in the preprocessing phase and 2ℓ bits in the online phase for each multiplication operation.

Inner product and convolution. Our maliciously secure inner product protocol Π_{Inner} is shown in Fig. 12. Its semi-honest version is the special case of Π_{PolyEvl} for 2-degree n-variate polynomial which requires one round communication of ℓ bits in the preprocessing phase and one round communication of 2ℓ bits in the online phase. To extend it to

Protocol $\Pi_{\mathsf{Inner}}(\langle x_1\rangle,\ldots,\langle x_n\rangle,\langle y_1\rangle,\ldots,\langle y_n\rangle)$

Input : $\langle\cdot\rangle$-shared value list of x_i and y_i.
Output : $\langle\cdot\rangle$-shared value of z where $z=\sum_{i=1}^{n} x_i\cdot y_i$.

Preprocessing:

- All parties prepare $[r_z] \leftarrow \Pi_{[\cdot]}$ locally;
- P_0 calculates $\Gamma=\sum_{i=1}^{n} r_{x_i}\cdot r_{y_i}+r_z$ and shares it with $\Pi_{[\cdot]}(\Gamma)$;

Online:

- P_j for $j\in\{1,2\}$ calculates $[m_z]_j = \sum_{i=1}^{n}(j-1)m_{x_i}\cdot m_{y_i} - m_{x_i}[r_{y_i}]_j - m_{y_i}[r_{x_i}]_j + [\Gamma]_j$ and mutually exchange their shares to reconstruct m_z.

Postprocessing:

- For N pairs inner product result $\{\{\langle x_i^{(j)}\rangle,\langle y_i^{(j)}\rangle\}_{i\in\mathbb{Z}_{n_j}};\langle z^{(j)}\rangle\}_{j\in\mathbb{Z}_N}$, all parties call $\Pi^{R}_{\mathsf{InnerVerify}}(\{\{\langle x_i^{(j)}\rangle,\langle y_i^{(j)}\rangle\}_{i\in\mathbb{Z}_{n_j}};\langle z^{(j)}\rangle\}_{j\in\mathbb{Z}_N})$ to verify correctness.

Figure 12: The Inner Product Protocol

the malicious setting, we employ batch verification protocol $\Pi^{R}_{\mathsf{InnerVerify}}$ (Fig. 11) to ensure the correctness of the inner products with a similar manner of multiplication. Analogously, in $\Pi^{R}_{\mathsf{InnerVerify}}$, 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 Π_{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 ℓ bits in the preprocessing phase and 2ℓ 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 Π_{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 ℓ_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 Π^{t}_{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 $r_z=\mathsf{rshift}(r_x,d)$, which allows the parties locally truncate $m_z=\mathsf{rshift}(m_x,d)$ in the online phase without communication. Although SWIFT [24] eliminates communication by combining truncation with multiplication, they still need 2ℓ online communication in the online phase of the standalone truncation protocol. Specifically, we let P_0 and P_1 pick random bit list $\{b_{1,j}\}_{j\in Z_\ell}$ together; P_0 and P_2 pick random bit list $\{b_{2,j}\}_{j\in Z_\ell}$ together. We utilize these lists to calculate that $r_x=\sum_{j=0}^{\ell-1}2^j\cdot(b_{1,j}\oplus b_{2,j})$ and $r_z=\sum_{j=0}^{\ell-t-1}2^j\cdot(b_{1,j}\oplus b_{2,j})+\sum_{j=\ell-t-1}^{\ell-1}2^j\cdot(b_{1,\ell-1}\oplus b_{2,\ell-1})$ which keeps the relationship $r_z=\mathsf{shift}(r_x,t)$. We can evaluate r_x and r_z under $\langle\cdot\rangle$-sharing to realize malicious security. To transform $b_{1,j}$ and $b_{2,j}$ to the $\langle\cdot\rangle$-sharing locally, we let $\langle b_{1,j}\rangle=(0,b_{1,j},0)$ and $\langle b_{2,j}\rangle=(0,0,b_{2,j})$ which set the other secret shard to be 0. For the result $\langle r_x\rangle$ and $\langle r_z\rangle$, since r_x and r_z is known by P_0, P_1 and P_2 can be locally calculate $[r_x]=m_{r_x}-[r_{r_x}]$ and $[r_z]=m_{r_z}-[r_{r_z}]$. Note that Π_{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 Π_{Trunc} requires 1 rounds and communication of 6ℓ 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

Protocol $\Pi^{t}_{\mathsf{Trunc}}(\langle x\rangle)$

Input : $\langle\cdot\rangle$-shared value.
Output : $\langle\cdot\rangle$-shared value of $z=\mathsf{rshift}(x,t)$.

Preprocessing:

- P_0 and P_i pick random bit list $\{b_{i,j}\}_{j\in Z_\ell}\leftarrow\mathbb{Z}_2^\ell$ together, for $i\in\{1,2\}$;
- All parties set
  - $\langle b_{1,j}\rangle := (m_{b_{1,j}},[r_{b_{1,j}}]_1,[r_{b_{1,j}}]_2) := (0,b_{1,j},0)$;
  - $\langle b_{2,j}\rangle := (m_{b_{2,j}},[r_{b_{2,j}}]_1,[r_{b_{2,j}}]_2) := (0,0,b_{2,j})$ for $j\in\mathbb{Z}_\ell$;
- All parties invoke Π_{Inner} to calculate
  - $\langle r_x\rangle=\sum_{j=0}^{\ell-1}2^j(\langle b_{1,j}\rangle+\langle b_{2,j}\rangle-2\langle b_{1,j}\rangle\cdot\langle b_{2,j}\rangle)$;
  - $\langle r_z\rangle=\sum_{j=0}^{\ell-t-1}2^j(\langle b_{1,j+t}\rangle+\langle b_{2,j+t}\rangle-2\langle b_{1,j+t}\rangle\cdot\langle b_{2,j+t}\rangle)+\sum_{j=\ell-t-1}^{\ell-1}2^j(\langle b_{1,\ell-1}\rangle+\langle b_{2,\ell-1}\rangle-2\langle b_{1,\ell-1}\rangle\cdot\langle b_{2,\ell-1}\rangle)$;
- P_0 set $r_x=\sum_{j=0}^{\ell-1}2^j\cdot(b_{1,j}\oplus b_{2,j})$, $r_z=\sum_{j=0}^{\ell-t-1}2^j\cdot(b_{1,j}\oplus b_{2,j})+\sum_{j=\ell-t-1}^{\ell-1}2^j\cdot(b_{1,\ell-1}\oplus b_{2,\ell-1})$;
- P_i for $i\in\{1,2\}$ set $[r_x]=m_{r_x}-[r_{r_x}]$, $[r_z]=m_{r_z}-[r_{r_z}]$;

Online:

- P_i for $i\in\{1,2\}$ set $m_z=\mathsf{rshift}(m_x,t)$;
- All parties output $\langle z\rangle := ([r_z],m_z)$.

Figure 13: The maliciously secure truncation protocol

ReLU. The ReLU of x is calculated by $w=x\cdot(1-\mathsf{sign}(x))=x-x\cdot\mathsf{sign}(x)$, which can be implemented by combining Π_{Mult} with Π_{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 Π_{SignBit}. We construct the semi-honest ReLU protocol Π_{ReLU} (Fig. 14) from Π_{SignBit}. Considering $\langle z\rangle=\Pi_{\mathsf{SignBit}}(\langle x\rangle)$ and $\langle w\rangle=\Pi_{\mathsf{Mult}}(\langle x\rangle\cdot\langle z\rangle)$, we have:

$$
\begin{aligned}
m_w &= m_x m_z + m_x r_z + m_z r_x + r_x r_z - r_w\\
&= m_x m_z + m_x r_z + (m' - 2\Delta m' + \Gamma)r_x + r_x r_z - r_w\\
&= m_x m_z + m_x r_z + (1-2\Delta)(m' r_x + r'') + \Gamma'
\end{aligned}
$$

Protocol $\Pi_{\mathsf{ReLU}}(\langle x \rangle)$

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

Preprocessing:

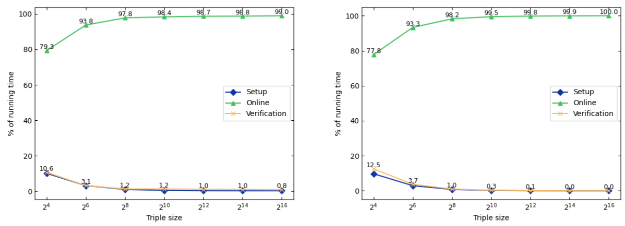
- All parties perform $[r''], [r'], [r_z], [r_w] \leftarrow \Pi_{[\cdot]}$;
- P_i, for $i \in \{1, 2\}$ pick $\Delta \in \{0, 1\}$ and reveal $[\Gamma] = \Delta + [r'] - 2\Delta \cdot [r'] + [r_z]$ to each other;
- P_i, for $i \in \{1, 2\}$ calculate $[\Gamma'] = \Gamma \cdot [r_x] - (1 - 2\Delta)[r''] + [r_x \cdot r_z] - [r_w]$;
- P_0 does:
  1) calculate $\hat{r_x} = -r_x - \mathsf{sign}(-r_x) \cdot 2^{\ell-1} \in \mathbb{Z}_{2^\ell}$;
  2) extract $2^{\ell-1} - 1 - \hat{r_x}$ as $\{r_{x,0}, \dots, r_{x,\ell-1}\}$;
  3) perform $[\![r_{x,j}]\!]^p \leftarrow \Pi^p_{[\![\cdot]\!]}(r_{x,j})$ for $j \in \mathbb{Z}^*_\ell$, taking the biggest prime of $p \in (\ell, 2^{\log \ell + 1}]$;
  4) perform $[r_x \cdot r_z] \leftarrow \Pi_{[\cdot]}(r_x \cdot r_z)$;

Online:

- P_i, for $i \in \{1, 2\}$ does:
  1) set $\hat{m_x} = m_x - \mathsf{sign}(m_x) \cdot 2^{\ell-1}$ and bitexact it as $\{\hat{m}_{x|j} \in \{0, 1\}\}_{j \in \mathbb{Z}_\ell}$ while $\sum_{j=0}^{\ell-1} 2^{\ell-1-j} \hat{m}_{x|j} = \hat{m_x}$;
  2) set $\hat{m}_{x|\ell} = 0$ and $[\![r_{x,\ell}]\!] = [\![1]\!]$;
  3) set $[\![m_j]\!]^p = \hat{m}_{x|j} + [\![r_{x,j}]\!]^p - 2\hat{m}_{x|j} \cdot [\![r_{x,j}]\!]^p$ for $j \in \mathbb{Z}^*_{\ell+1}$.
  4) pick same random values $\{w_j, w'_j \in \mathbb{Z}^*_p\}_{j \in \mathbb{Z}^*_{\ell+1}}$ via PRF with seed η;
  5) calculate $[\![m'_j]\!]^p = \sum_{k=1}^{j} [\![m_k]\!]^p - 2 \cdot [\![m_j]\!]^p + 1$ and $[\![u_j]\!]^p = w_j \cdot [\![m'_j]\!]^p + (\mathsf{sign}(m_x) \oplus \hat{m}_{x|j} \oplus \Delta)$ and $[\![u'_j]\!]^p = w'_j(w_j \cdot [\![m'_j]\!]^p + 1)$ for $j \in \mathbb{Z}^*_{\ell+1}$;
  6) pick a random permutation π via PRF with seed η and permute the list $\{[\![\hat{u_j}]\!]^p\}_{j \in \mathbb{Z}^*_{\ell+1}} = \pi(\{[\![u_j]\!]^p\}_{j \in \mathbb{Z}^*_{\ell+1}})$ and $\{[\![\hat{u'_j}]\!]^p\}_{j \in \mathbb{Z}^*_{\ell+1}} = \pi(\{[\![u'_j]\!]^p\}_{j \in \mathbb{Z}^*_{\ell+1}})$;
  7) reveal $\{[\![\hat{u_j}]\!]^p\}_{j \in \mathbb{Z}^*_{\ell+1}}$ and $\{[\![\hat{u'_j}]\!]^p\}_{j \in \mathbb{Z}^*_{\ell+1}}$ to P_0 and reveal $\Gamma'' = m_x \cdot [r_z] + [\Gamma']$ to each other simultaneously;
- P_0 sets $m' = \mathsf{sign}(-r_x) - r'$ if $\exists \hat{u_j} = 0 \wedge \hat{u'_j} \neq 0$ for $j \in \mathbb{Z}^*_{\ell+1}$ else $m' = (1 \oplus \mathsf{sign}(-r_x)) - r'$;
- P_0 sets $m'' = m' \cdot r_x + r''$;
- P_0 sends m' and m'' to P_i, for $i \in \{1, 2\}$;
- P_i, for $i \in \{1, 2\}$ sets $m_z = m' - 2\Delta \cdot m' + \Gamma$ and $m_w = m_x m_z + (1 - 2\Delta)m'' + \Gamma''$;
- All parties output $\langle z \rangle := ([r_z], m_z)$ and $\langle w \rangle := ([r_w], m_w)$.

Figure 14: The 2-round ReLU Protocol.

m', Δ, Γ are the fresh random values mentioned in Π_{SignBit} and it hold $m_z = m' - 2\Delta m' + \Gamma$ in Π_{SignBit}. We denote $\Gamma' = \Gamma \cdot r_x - (1 - 2\Delta)r'' + r_x \cdot r_z - r_w$, where r'' is a fresh random introduced to protect the privacy of r_w. We let P_1 and P_2 calculate $[\Gamma'] = \Gamma \cdot [r_x] - (1 - 2\Delta)[r''] + [r_x \cdot r_z] - [r_w]$ locally in the offline phase. P_1 and P_2 reveal $[\Gamma''] = m_x \cdot [r_z] + [\Gamma']$ to each other in the first round of Π_{SignBit}. For item $(1 - 2\Delta)(m' r_x + r'')$, P_0 send $m'' = m' r_x + r''$ to P_1 and P_2. Then P_1, P_2 locally calculate $m_w = m_x \cdot m_z + \Gamma'' + (1 - 2\Delta)m''$. Note that reveal m'' and Γ'' will not leak any information, since the P_1 and P_2 cannot extract additional information of r_x, r_z, 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.



(a) LAN (b) MAN

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

The malicious version of ReLU can be achieved through verifying $\langle z \rangle = \mathsf{sign}(\langle x \rangle)$ and $\langle w \rangle = \Pi_{\mathsf{Mult}}(\langle x \rangle, \langle z \rangle)$ respectively.

Security. We analyze the security of our ReLU protocol in the UC framework. The functionality $\mathcal{F}_{\mathsf{ReLU}}$ is defined as an instantiation of $\mathcal{F}_{\mathsf{3pc}}$ depicted in Fig. 2; namely, it calculates $w = \mathsf{ReLU}(x)$ and $z = \mathsf{sign}(x)$.

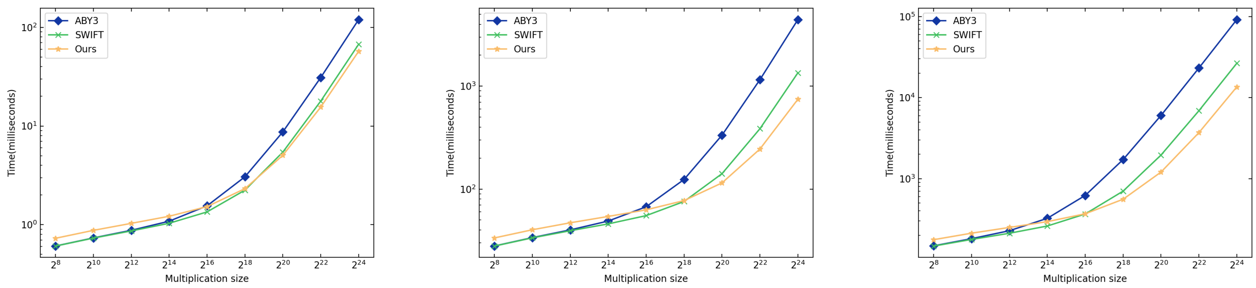
Theorem 4. *Let $\mathsf{PRF}^{(\mathbb{Z}_p)^p}$, $\mathsf{PRF}^{\mathbb{Z}_p}$ and $\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}$ be the secure pseudo-random functions. The protocol Π_{ReLU} depicted in Fig. 8 UC realizes $\mathcal{F}_{\mathsf{ReLU}}$ against semi-honest PPT adversaries who can statically corrupt up to one party.*

Proof. See Appendix A.7. □

Maxpool. Our Maxpool scheme is constructed by the comparison function $\mathsf{great}(x, y) = x \overset{?}{>} y$ and the maximum function $\max(x_1, \dots, x_n)$. In the case of signed numbers x and y, $\mathsf{great}(x, y)$ can be implemented by invoking the Π_{VSignBit} three times. That is, $\mathsf{great}(x, y) = (\mathsf{sign}(x) \oplus \mathsf{sign}(y)) \cdot \mathsf{sign}(y - x) + (1 \oplus \mathsf{sign}(x) \oplus \mathsf{sign}(y)) \cdot \mathsf{sign}(y)$. For unsigned number x and y which $\mathsf{sign}(x) = 0$ and $\mathsf{sign}(y) = 0$, we have $\mathsf{great}(x, y) = \mathsf{sign}(y - x)$. 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 $\mathsf{sign}(y - x)$.

There are two approaches to evaluate $\max(x_1, \dots, x_n)$. One is to evaluate $\max(x_1, \dots, x_n)$ by $\max(x_1, \dots, x_n) = \sum_{i=1}^{n} (\Pi_{j=1, j \neq i}^{n} \mathsf{great}(x_i, x_j) \cdot x_i)$, 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 expending $\log(n)$ times $\max(x_1, \dots, x_n) = \max(\max(x_1, x_2), \dots, v(x_{n-1}, x_n))$. 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 $z_{i,j}$ is the result element of performing $(2, 2)$-kernel shape and 1-stride Maxpool over an $a \times b$-dimension matrix requires where $z_{i,j} = \max(x_{i,j}, x_{i,j+1}, x_{i+1,j}, x_{i+1,j+1})$ and



(a) LAN. (b) MAN (c) WAN

Figure 16: Overall run-time of multiplication. Comparison with ABY3 [27], SWIFT [24] of Π_{Mult} over LAN, MAN and WAN. The multiplication depth is logarithmic of the multiplication size.

$z_{i,j+1} = \mathsf{max}(x_{i,j+1}, x_{i,j+2}, x_{i+1,j+1}, x_{i+1,j+2})$. Both $z_{i,j}$ and $z_{i,j+1}$ needs the outcome of $\mathsf{great}(x_{i,j+1}, x_{i+1,j+1})$. 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 $\mathsf{max}(x_i, x_j)$ can be calculated as $(x_i - x_j) \cdot \mathsf{great}(x_i, x_j) + x_j$, i.e. $(x_i - x_j) \cdot \mathsf{sign}(x_j - x_i) + x_j$. In the previous chapter, we implemented $f(x) = x \cdot \mathsf{sign}(x)$ in two rounds by introducing 2ℓ bits of communication overhead in the online phase. We use it to evaluate $\mathsf{max}(x_i, x_j)$ by $\mathsf{max}(x_i, x_j) = x_j - f(x_j - x_i)$. We apply this approach to evaluate Maxpool, which requires $(n-1)(\ell \log \ell + 4\ell)$ 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 Bicoptor [40], BLAZE [30], SWIFT, and ABY respectively.

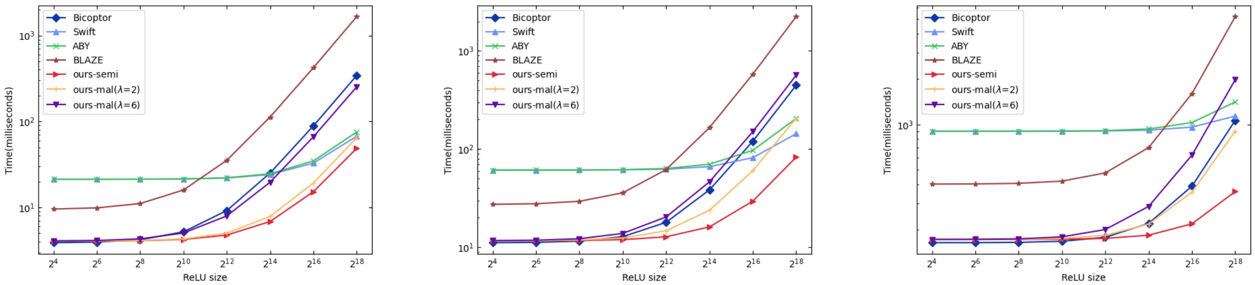
Benchmark setting. We take the size of the ring $\ell = 64$ and the polynomial ring degree $d = 32$. For the fixed-point value, we utilize 16 bits truncation. Our experiments are performed in a local area network, using software to simulate three network settings: local-area network (LAN, RTT: 0.2ms, 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.

Multiplication. We compare our maliciously secure multiplication protocol with SOTA. We benchmark the communication of Π_{Mult} and Π_{Inner} in the Appendix C.1. Fig. 16 depicts the running time of Π_{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.

Trade-off of the repetition parameter R. While selecting a larger value for the repetition parameter R for dimension reduction can minimize the communication volume in batch verification, it is also essential to consider the impact of additional communication rounds in the postprocessing phase for overall performance. We conduct a practical experimental benchmark to determine the optimal value of R in different bandwidth and delay scenarios. By simulating various bandwidths and delays, we can identify the most suitable value of R for each specific scenario. We describe the procedure for selecting R in Appendix C.4.

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



(a) LAN (b) MAN (c) WAN

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].

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 $4\times$ compared to the constant round protocol Bicoptor (theoretically, communication volume has been reduced by $4\times$ on a 64-bit ring). Under the malicious threat model, compared to the constant round protocol BLAZE, our maliciously secure version achieves over $10\times$ 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 $4\times$ 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 28×28 image and involves a convolution layer(5 kernels with 5×5 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 32×32 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 64×64 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. (Com: the communication which is given in MB. Time: the run-time which is given in ms)

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

TABLE 3 depicts the run-time and communication of our protocol under the LAN setting. Our benchmark contains the communication cost and the running time of each stage. In the execution stage, all parties perform offline/online procedures of the semi-honest protocols. In the verification stage, all parties perform a postprocessing procedure to verify the correctness of the shared result. Our platform can execute CNNs-like LeNet in hundreds of milliseconds. For the deeper CNNs such as VGG, our platform can complete the execution within tens of seconds.

8. Conclusion

We propose Aegis, an efficient PPML framework that achieves malicious security in an honest majority. We apply the batch multiplication verification protocol on the 3PC over the ring. We innovate novel semi-honest and maliciously secure sign-bit extraction protocols. We then expand the sign-bit extraction protocol to applications such as

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

[1] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, 1991.

[2] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT*, 2011.

[3] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In *CRYPTO*, 2019.

[4] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In *CCS*, 2019.

[5] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Sublinear gmw-style compiler for mpc with preprocessing. In *Advances in Cryptology – CRYPTO 2021*, 2021.

[6] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. Flash: Fast and robust framework for privacy-preserving machine learning. In *PoPETs*, 2020.

[7] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

[8] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. Astra: High throughput 3pc over rings with application to secure prediction. In *CCSW*, 2019.

[9] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4pc framework for privacy preserving machine learning. In *NDSS*, 2020.

[10] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority mpc for malicious adversaries. In *CRYPTO*, 2018.

[11] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority mpc for malicious adversaries. In *CRYPTO*, 2018.

[12] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. Spdz2k: Efficient MPC mod 2k for dishonest majority. In *CRYPTO*, 2018.

[13] Anders Dalskov, Daniel Escudero, and Ariel Nof. Fast fully secure multi-party computation over any ring with two-thirds honest majority. In *CCS*, 2022.

[14] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, 2012.

[15] Ivan Damgård, Claudio Orlandi, and Mark Simkin. Yet another compiler for active security or: Efficient mpc over arbitrary rings. In *CRYPTO*, 2018.

[16] Daniel Escudero and Vipul Goyal. Turbopack: Honest majority mpc with constant online communication. In *CCS*, 2022.

[17] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*, 1987.

[18] Vipul Goyal, Hanjun Li, Rafail Ostrovsky, and Antigoni Polychroniadou. Atlas: Efficient and scalable mpc in the honest majority setting. In *CRYPTO*, 2021.

[19] Vipul Goyal and Yifan Song. Malicious security comes free in honest-majority mpc. Cryptology ePrint Archive, Paper 2020/134, 2020.

[20] Carmit Hazay, Abhi Shelat, and Muthuramakrishnan Venkitasubramaniam. Going beyond dual execution: Mpc for functions with efficient verification. In *PKC*, 2020.

[21] Eerikson Hendrik, Keller Marcel, Orlandi Claudio, Pullonen Pille, Puura Joonas, and Simkin Mark. Use your brain! arithmetic 3pc for any modulus with active security. In *ITC*, 2020.

[22] Wilko Henecka, Stefan K ögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Tasty: Tool for automating secure two-party computations. In *CCS*, 2010.

[23] Yan Huang, Jonathan Katz, and David Evans. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *S&P*, 2012.

[24] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. Swift: Super-fast and robust privacy-preserving machine learning. In *USENIX*, 2021.

[25] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow: Secure tensorflow inference. In *S&P*, 2020.

[26] Eleftheria Makri, Dragos Rotaru, Nigel P. Smart, and Frederik Vercauteren. Epic: Efficient private image classification (or: Learning from the masters). In *CT-RSA*, 2019.

[27] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *CCS*, 2018.

[28] Peter Sebastian Nordholt and Meilof Veeningen. Minimising communication in honest-majority mpc by batchwise multiplication verification. In *ACNS*, 2018.

[29] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. Aby2. 0: Improved mixed-protocol secure two-party computation. In *USENIX*, 2021.

[30] Arpita Patra and Ajith Suresh. BLAZE: blazing fast privacy-preserving machine learning. In *NDSS*, 2020.

[31] Mohassel Payman and Zhang Yupeng. Secureml: A system for scalable privacy-preserving machine learning. In *S&P*, 2017.

[32] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow2: Practical 2-party secure inference. In *CCS*, 2020.

[33] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *ASIACCS*, 2018.

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

[35] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. In *PoPETs*, 2021.

[36] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. Piranha: A gpu platform for secure computation, 2022.

[37] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. Piranha source code, 2022.

[38] Andrew C. Yao. Protocols for secure computations. In *SFCS*, 1982.

[39] Lindell Yehuda and Nof Ariel. A framework for constructing fast mpc over arithmetic circuits with malicious adversaries and an honest-majority. In *CCS*, 2017.

[40] Lijing Zhou, Ziyu Wang, Hongrui Cui, Qingrui Song, and Yu Yu. Bicoptor: Two-round secure three-party non-linear computation without preprocessing for privacy-preserving machine learning. In *S&P*, 2023.

Appendix A. Security Proofs

A.1. The proof of Theorem 1.

Theorem 1. *Let* $\mathsf{PRF}^{(\mathbb{Z}_p)^p}$,$\mathsf{PRF}^{\mathbb{Z}_p}$ *and* $\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}$ *be the secure pesudo-random functions. The protocol* Π_{SignBit} *as depicted* *in Fig. 3 UC realizes $\mathcal{F}_{\mathsf{SignBit}}$ against semi-honest PPT adversaries who can statically corrupt up to one party.*

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

Case 1: P_0 is corrupted.

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

- Upon receiving $\{[\![r_{x,j}]\!]_1^p\}_{j\in\mathbb{Z}_\ell^*}, [r']_1$ form corrupted P_0 to P_1, and $\{[\![r_{x,j}]\!]_2^p\}_{j\in\mathbb{Z}_\ell^*}, [r']_2$ form corrupted P_0 to P_2, $\mathcal{S}$ extracts $\hat{r_x} = 2^{\ell-1} - 1 - \sum_{j=1}^{\ell-1}([\![r_{x,j}]\!]_1^p + [\![r_{x,j}]\!]_2^p)$ and $r' = [r']_1 + [r']_2$;
- $\mathcal{S}$ picks random list $\{\hat{u'_j}\}_{j\in\mathbb{Z}_{\ell+1}^*}$ where $\hat{u'_j} \in \mathbb{Z}_p$ and sets another list $\{\hat{u}_j\}_{j\in\mathbb{Z}_{\ell+1}^*}$ as following steps:
  - For each j where $\hat{u'_j} = 0$, set $\hat{u}_j \leftarrow \{p-1, 0\}$.
  - For each j where $\hat{u'_j} \neq 0$, set $\hat{u}_j \leftarrow \mathbb{Z}_p^*$.
  - If $\exists j$ such that $\hat{u'_j} \neq 0$, select random one of j to set $\hat{u}_j \leftarrow \mathbb{Z}_2$.
  - sends $\{\hat{u'_j}\}_{j\in\mathbb{Z}_{\ell+1}^*}$ and $\{\hat{u'_j}\}_{j\in\mathbb{Z}_{\ell+1}^*}$ to P_0.
- Upon receiving m' from corrupted P_0 to P_1 and P_2, $\mathcal{S}$ does:
  - If $\exists \hat{u}_j = 0 \wedge \hat{u'_j} \neq 0$, set $\mathsf{sign}(-r_x) = (m' + r')$, else set $\mathsf{sign}(-r_x) = (m' + r') \oplus 1$.
  - Calculate $r_x = -\hat{r_x} - \mathsf{sign}(-r_x) \cdot 2^{\ell-1}$.
  - Send (Input, r_x) to $\mathcal{F}_{\mathsf{SignBit}}$.
  - Generate $[r_z]_1$, $[r_z]_2$ with seed and send to $\mathcal{F}_{\mathsf{SignBit}}$.

Indistinguishability. The indistinguishability is proven through a series of hybrid worlds $\mathcal{H}_0, \mathcal{H}_1$.

Hybrid $\mathcal{H}_0$: It is the real protocol execution $\mathsf{Real}_{\Pi_{\mathsf{SignBit}},\mathcal{A},\mathcal{Z}}(1^\lambda)$.

Hybrid $\mathcal{H}_1$: It is same as $\mathcal{H}_0$ except that in $\mathcal{H}_1$, list $\hat{u}_j$ and $\hat{u'_j}$ are picked uniformly random instead of calculating from $w_j \cdot m'_j + (\mathsf{sign}(m_x) \oplus \hat{m_{x|j}} \oplus \Delta)$ and $w'_j(w_j \cdot m'_j + 1)$.

Claim 1. *If $\mathsf{PRF}^{\mathbb{Z}_p}$ and $\mathsf{PRF}^{(\mathbb{Z}_p)^p}$ are the secure pseudorandom functions with adversarial advantage $\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_p}}(1^\lambda, \mathcal{A})$ and $\mathsf{Adv}_{\mathsf{PRF}^{(\mathbb{Z}_p)^p}}(1^\lambda, \mathcal{A})$, then $\mathcal{H}_1$ and $\mathcal{H}_0$ are indistinguishable with advantage $\epsilon < 2 \cdot \ell \cdot \mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_p}}(1^\lambda, \mathcal{A}) + \mathsf{Adv}_{\mathsf{PRF}^{(\mathbb{Z}_p)^p}}(1^\lambda, \mathcal{A})$.*

Case 2: P_1(or P_2) is corrupted.

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

- $\mathcal{S}$ generate Δ, $[r_z]_1$ with the seed and sends $[r_z]_1$ to $\mathcal{F}_{\mathsf{SignBit}}$.
- $\mathcal{S}$ picks $[\![r_{x,j}]\!]_1 \leftarrow \mathbb{Z}_p$ and acts as P_0 to send it to P_1.
- $\mathcal{S}$ picks $[\Gamma]_2 \leftarrow \mathbb{Z}_{2^\ell}$ and acts as P_2 to send it to P_1.
- Upon receiving $[\Gamma]_1$ from P_1, $\mathcal{S}$ calculates $\Gamma = [\Gamma]_1 + [\Gamma]_2$.
- Upon receiving $\{[\![\hat{u}_j]\!]_1\}_{j\in\mathbb{Z}_{\ell+1}^*}$ and $\{[\![\hat{u'_j}]\!]_1\}_{j\in\mathbb{Z}_{\ell+1}^*}$ from corrupted P_1 to P_0, $\mathcal{S}$ does.
  - invoke PRF with η to generate permutation π.
  - invoke PRF with η to generate $w_j, w'_j \in \mathbb{Z}_p^*$ for $j \in \mathbb{Z}_{\ell+1}^*$.
  - invoke PRF with η to generate $\Delta \in \mathbb{Z}_2$.
  - calculate $\{[\![u_j]\!]_1\}_{j\in\mathbb{Z}_{\ell+1}^*} = \pi^-(\{[\![\hat{u}_j]\!]_1\}_{j\in\mathbb{Z}*_{\ell+1}})$ and $\{[\![u'_j]\!]_1\}_{j\in\mathbb{Z}_{\ell+1}^*} = \pi^-(\{[\![\hat{u'_j}]\!]_1\}_{j\in\mathbb{Z}*_{\ell+1}})$.
  - calculate $\hat{m_{x|j}}$ via $\{[\![u'_j]\!]_1\}_{j\in\mathbb{Z}_{\ell+1}^*}$, w_j, w'_j and $[\![r_{x,j}]\!]_1$.
  - calculate $\mathsf{sign}(m_x)$ via Δ, $\hat{m_{x|j}}$ and $\{[\![u_j]\!]_1\}_{j\in\mathbb{Z}_{\ell+1}^*}$.
  - calculate $m_x = \mathsf{sign}(m_x) \cdot 2^{\ell-1} + \sum_{j=1}^{\ell} 2^{\ell-j-1} \cdot \hat{m_{x|j}}$
- $\mathcal{S}$ sends (Input, m_x) to $\mathcal{F}_{\mathsf{SignBit}}$ and receives $(\mathsf{Output}, m_z, [r_z]_1)$.
- $\mathcal{S}$ acts as P_0 to send $m' = (m_z - \Gamma)/(1 - 2\Delta)$ to P_1.

Indistinguishability. The indistinguishability is proven through a series of hybrid worlds $\mathcal{H}_0, \mathcal{H}_1$.

Hybrid $\mathcal{H}_0$: It is the real protocol execution $\mathsf{Real}_{\Pi_{\mathsf{SignBit}},\mathcal{A},\mathcal{Z}}(1^\lambda)$.

Hybrid $\mathcal{H}_1$: It is same as $\mathcal{H}_0$ except that in $\mathcal{H}_1$, $[\![r_{x,j}]\!]_1$, $[\Gamma]_1$ and m' are picked uniformly random instead of calculating from $r_{x,j}$, $\Delta + [r']_2 - 2\Delta \cdot [r']_2 + [r_z]_2$ and $\mathsf{sign}(-r_x) - r'$.

Claim 2. *If $\mathsf{PRF}^{\mathbb{Z}_p}$ and $\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}$ are the secure pseudorandom functions with adversarial advantage $\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_p}}(1^\lambda, \mathcal{A})$ and $\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}}(1^\lambda, \mathcal{A})$, then $\mathcal{H}_1$ and $\mathcal{H}_0$ are indistinguishable with advantage $\epsilon = \ell \cdot \mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_p}}(1^\lambda, \mathcal{A}) + 2\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}}(1^\lambda, \mathcal{A})$.*

Proof. We replace the ℓ $\mathsf{PRF}^{\mathbb{Z}_p}$ outputs and 2 $\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}$ outputs to uniformly random number; therefore,the overall advantage is $\epsilon = \ell \cdot \mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_p}}(1^\lambda, \mathcal{A}) + 2\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}}(1^\lambda, \mathcal{A})$ by hybrid argument via reduction. $\square$

This concludes the proof. $\square$

A.2. The proof of Lemma 1.

Lemma 1. *Suppose protocol Π_{Trans} depicted in Fig. 4 take input as $\{\langle x_i\rangle, \langle y_i\rangle, \langle z_i\rangle\}_{i\in\mathbb{Z}_N}$, and it outputs $\{\langle x'_i\rangle^{\ell[x]}, \langle y_i\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_N}; \langle z\rangle^{\ell[x]}$. 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. $\mathbb{Z}_{2^\ell}[x]/f(x)$:*

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

Proof. It is sufficient to show that r is uniformly random if Π_{Rec} is not abort. The adversary tries to make $\sum_{i=0}^{N} r^{i-1} \cdot z_i = \sum_{i=0}^{N} r^{i-1} \cdot x_i \cdot y_i$ 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 $\sum_{i=0}^{N} r^{i-1} \cdot x_i \cdot y_i = \sum_{i=0}^{N} r^{i-1} \cdot (x_i \cdot y_i + e_i)$. The condition that makes the equation hold is the random value r is the root of $f(x) = \sum_{i=0}^{N} x^{i-1} \cdot e_i$. Since the size of roots of $N-1$-degree $f(x)$ over $\mathbb{Z}_{2^\ell}[x]$ is $2^{(\ell-1)d}N + 1$, the probablity that uniformly random value r match the root is $\frac{2^{(\ell-1)d}N+1}{2^{\ell d}} \approx \frac{N}{2^d}$. $\square$

A.3. The proof of Lemma 2.

Lemma 2. *Suppose protocol Π_{Reduce} depicted in Fig. 5 take input as $(\{\langle x_i\rangle^{\ell[x]}, \langle y_i\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_N}, \langle z\rangle^{\ell[x]})$, and it outputs $(\{\langle x_i'\rangle^{\ell[x]}, \langle y_i'\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_{N/2}}, \langle z'\rangle^{\ell[x]})$. The probability that the following two conditions hold is at most $\frac{1}{2^{d-1}}$, where d is the degree of $f(x)$ w.r.t. $\mathbb{Z}_{2^\ell}[x]/f(x)$:*

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

Proof. For the convenience of description, we denote $h'(k) = \sum_{i=0}^{N/2} f_i(k)\cdot g_i(k)$. The adversary tries to make $h(\zeta) = h'(\zeta)$ 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 introduce 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 $h(\zeta) = \sum_{i=0}^{2}((\Pi_{j=1,j\neq i}^{2}\frac{\zeta-j}{i-j})\cdot h(i)) = \frac{(\zeta-1)\cdot(\zeta-2)}{2}\cdot h(0) + \zeta\cdot(2-\zeta)\cdot h(1) + \frac{(\zeta-1)*\zeta}{2}\cdot h(2)$, to make it equal to $h'(\zeta) = \frac{(\zeta-1)\cdot(\zeta-2)}{2}\cdot h'(0) + \zeta\cdot(2-\zeta)\cdot h'(1) + \frac{(\zeta-1)*\zeta}{2}\cdot h'(2)$, is to make $\frac{(\zeta-1)\cdot(\zeta-2)}{2}\cdot e_1 + \zeta\cdot(2-\zeta)\cdot(e-e_1) + \frac{(\zeta-1)\cdot\zeta}{2}\cdot(e_2) = 0$ for random picked $\zeta\in\mathbb{Z}_{2^\ell}[x]$. The probability that the adversary deliberately chooses e, e_1, e_2 to make the equation hold is to make ζ be the root of 2-degree polynomial $f(x) = \frac{(x-1)\cdot(x-2)}{2}\cdot e_1 + x\cdot(2-x)\cdot(e-e_1) + \frac{(x-1)\cdot x}{2}\cdot(e_2)$ over $\mathbb{Z}_{2^\ell}[x]$, which is at most $2^{2(\ell-1)d}+1$. So we have the soundness error $\frac{2^{(\ell-1)d+1}+1}{2^{\ell d}} \approx \frac{1}{2^{d-1}}$ □

A.4. The proof of Lemma 3.

Lemma 3. *Let $(\{\langle x_i\rangle^{\ell[x]}, \langle y_i\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_N}, \langle z\rangle^{\ell[x]})$ be the input of protocol $\Pi_{\mathsf{InnerVerify}}$ depicted in Fig. 6. The probability that $\Pi_{\mathsf{InnerVerify}}$ 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. $\mathbb{Z}_{2^\ell}[x]/f(x)$.*

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

A.5. The proof of Theorem 2.

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

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

A.6. The proof of Theorem 3.

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

Proof. For each illegel u_j in $\Pi^\lambda_{\mathsf{Pos}}$, the probability that malicious P_i for $i\in\{1,2\}$ make it pass the MAC check is $\frac{1}{2^{(\log\ell+1)\lambda}}$ w.r.t. the MAC key space $\mathbb{Z}_p^\lambda$ (taking $p\approx 2^{(\log\ell+1)}$). 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 $\frac{1}{\ell}$. So the soundness error is $\frac{1}{2^{(\log\ell+1)\lambda}\ell} = \frac{1}{2^{\lambda\log\ell+\lambda+\log\ell}}$. □

A.7. The proof of Theorem 4.

Theorem 4. *Let $\mathsf{PRF}^{(\mathbb{Z}_p)^p}$,$\mathsf{PRF}^{\mathbb{Z}_p}$ and $\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}$ be the secure pesudo-random functions. Let $\ell = \mathsf{poly}(\lambda)$. The protocol Π_{ReLU} depicted in Fig. 8 UC realizes $\mathcal{F}_{\mathsf{SignBit}}$ 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 Π_{SignBit}, Π_{ReLU} sends Γ'', m'' to corrupted P_1 (or P_2) and introduce the PRF outputs r'', r_w to make Γ'', m'' uniformly random, where make the indistinguishable advantage of corrupted P_1 equal to $\epsilon = \ell\cdot\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_p}}(1^\lambda,\mathcal{A}) + 2\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}}(1^\lambda,\mathcal{A})$ □

Appendix B. Other semi-honest protocol

B.1. Semi-honest secure truncation

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

Protocol $\Pi^t_{\mathsf{semi\text{-}trunc}}(\langle x\rangle)$

Input: $\langle\cdot\rangle$-shared value.
Output: $\langle\cdot\rangle$-shared value of $z = \mathsf{rshift}(x,t)$.

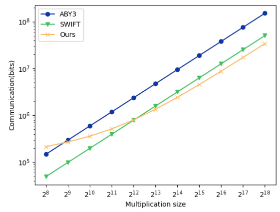
Preprocessing:

- P_0 pick random value r_x which satisfy $\mathsf{rshift}(r_x,t) = \mathsf{rshift}([r_x]_1,t) + \mathsf{rshift}([r_x]_2,t)$.
- All parties perform $[r_x]\leftarrow\Pi_{[\cdot]}(r_x)$.
- All parties set $[r_z]_i = \mathsf{rshift}([r_x]_i,t)$ for $i\in\{1,2\}$

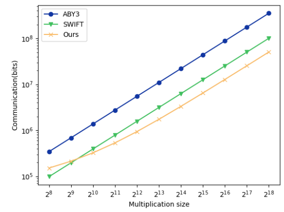
Online:

- P_i for $i\in\{1,2\}$ set $m_z = \mathsf{rshift}(m_x,t)$
- All parties output $\langle r_z\rangle = ([r_z], m_z)$

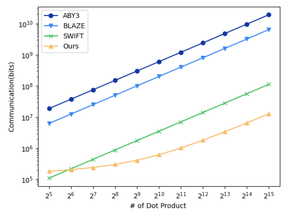
Figure 18: The semi-honest truncation protocol



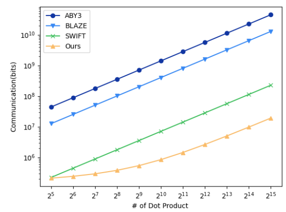
(a) Online com. of MUL



(b) Total com. of MUL



(c) Online communication of Inner Product with Trunction



(d) Overall communication of Inner Product with Trunction

Figure 19: Communication overhead comparison with ABY3 [27], BLAZE [30], SWIFT [24] of muliplication 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 Π_{Reduce} to reduce dimensionality to 1, and the scheme that directly employs $\Pi_{\mathsf{InnerVerify}}$. 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 Π_{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

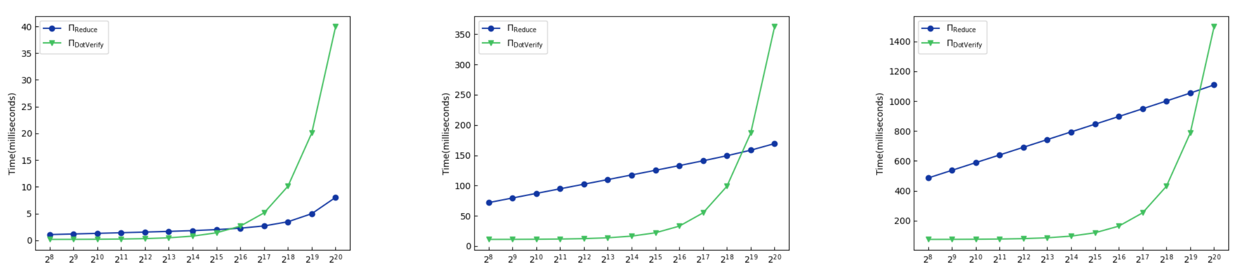
In the honest-majority setting, several works such as [11], [13], [15], [16], [18], [39] have designed protocols for efficient secure multi-party computation against the malicious adversary. However, compared to the semi-honest case, previous work requires significantly higher additional overhead. For 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 $1\frac{1}{3}$ ring elements with two rounds of communication or $1\frac{2}{3}$ 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], [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.

Operation	Input Size	Communication		Time.(ms)		Throughput. (ops/s)
		Offline	Online	Offline	Online	
Sign	2^4	1.1 KB	4.2 KB	11.52	19.41	516
	2^8	16.6 KB	66.4KB	11.96	19.99	8050
	2^{16}	4.2MB	17.0MB	77.59	249.58	200415
ReLU	2^4	1.3KB	5.2KB	11.67	19.47	513
	2^8	20.7KB	83.1KB	11.96	20.01	8007
	2^{16}	5.3MB	21.2MB	77.71	262.12	192849
MaxPool	2^4	1.1KB	5.1KB	11.75	36.38	333
	2^8	20.6KB	82.8KB	11.86	73.28	3006
	2^{16}	5.3MB	21.2MB	76.04	564.42	102326

TABLE 5: The communication cost of our protocols. (Offline.Com./Online.Com./Com.: the communication cost of offline/online/verification phase. Rounds: the communication rounds of the online phase. ℓ is the ring size. λ:the statistical security parameter. n:the MaxPool size. R:the dimension reduction times. N:the data size. M:the inner product dimension.)

Operation	Execution(Semi-honest)			Verification	
	Offline.Com.(bit)	Rounds	Online.Com.(bit)	Rounds	Com.(bit)
Multiplication	ℓ	1	2ℓ	$R+1$	$(11R+3N/2^R+9)\ell \cdot d$
Inner Product	ℓ	1	2ℓ	$R+1$	$(11R+3N \cdot M/2^R+9)\ell \cdot d$
Truncation	ℓ	0	0	$R+1$	$(11R+6N/2^R+9)\ell \cdot d$
Sign-bit Extraction	$\ell(1+\log\ell)+2\ell$	2	$4\ell(\log\ell+1)+2\ell$	2	$10\lambda\ell(\log\ell+1)+14\ell\log\ell+16\ell$
ReLU	$\ell(1+\log\ell)+3\ell$	2	$4\ell(\log\ell+1)+4\ell$	2	$10\lambda\ell(\log\ell+1)+14\ell\log\ell+18\ell$
MaxPool	$(n-1)(4\ell+\ell\log\ell)$	$\log n$	$(n-1)4\ell(\log\ell+2)$	$\log n$	$(n-1)\ell((10\lambda+14)(\log\ell+1)+4\ell)$



(a) LAN. (b) MAN (c) WAN

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 ℓ is small. On the other hand, [40] implements comparison through a truncation protocol. Their approach performs local truncation ℓ times, followed by involving a third party to verify if the result contains zero items. This scheme realizes two rounds of ℓ^2 bits communication. However, this approach has not been applied to malicious threat models.