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); communication and round complexity of its malicious version are approximately 25% and 50% of the SOTA (BLAZE) by Patra and Suresh (NDSS 2020), for $\ell = 64$. Moreover, the overall 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 constructions by $4\times$ in the semi-honest setting and $100\times$ 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) [2], [20], [41] allows n parties to jointly evaluate certain functions without revealing their private inputs, and it is a typical cryptographic tool to realize PPML [8], [29], [30], [33], [36], [38] in the multi-server setting. (This work focuses on 3-party MPC, denoted as 3-PC.) Most of these protocols [10], [37] 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 [16], [27] is at least $2\times$ slower than its semi-honest version.

PPML-friendly MPC protocols usually operate over a finite ring $\mathbb{Z}_{2^\ell}$ 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 $\mathbb{Z}_p$. Recently, there has been a series of works [19], [22], [31], implementing efficient maliciously secure protocols over $\mathbb{Z}_p$. Certain 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 [14], [18], [24] have been made to transform those techniques to MPC over $\mathbb{Z}_{2^\ell}$, but the resulting maliciously secure protocols come with a $2\times$ communication overhead. Alternatively, another line of work [16], [27], [33] tries to design maliciously secure MPC over $\mathbb{Z}_{2^\ell}$ from scratch. However, 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 approach [11], [25], [30], [34] 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 method needs costly share conversion between arithmetic and boolean fields, which typically requires logarithmic communication rounds w.r.t. the share length. Recently, many SOTA PPML protocols [5], [28], [37], [38], [43] propose tailor-made protocols to evaluate certain non-arithmetic functions, e.g., comparison and ReLU, eliminating the need for share conversion. However, Falcon [38] still needs logarithmic communication rounds; SecureNN [37] and CrypTFlow [28] require more than 8 rounds of communication, and in most cases, it is even more than logarithmic rounds ($\ell > 32$, ABY^3 [30]). Boyle *et al.* [5] introduce the function secret sharing (FSS) scheme to perform comparison. It only requires one round of communication in the online phase, as a sacrifice, it introduces massive computation and offline communication, which is $O(\kappa\ell)$ where the security parameter $\kappa = 128$. Our experiments (Cf. Sec. 6) show that the performance of FSS in most practical scenarios is far worse than other schemes in terms of overall running time. Recently, Zhou *et al.* [43]

proposes a novel 2-round comparison protocol based on probabilistic truncation, and it costs $O(\ell^2)$ communication. Nevertheless, when encountering a large ring size ℓ, it performs even worse than FSS. To the best of our knowledge, there is no efficient constant-round 3-PC protocol with low communication for non-linear function evaluations.

Our results. In this work, we propose Aegis – a maliciously secure PPML platform that is based on 3-party MPC in the honest majority setting. The underlying share of our 3-PC protocol originates from a variant of the replicated secure sharing (RSS) [10]; that is, to share $x \in \mathbb{Z}_{2^\ell}$, P_0 holds (r_1, r_2), P_1 holds $(m = x - r, r_1)$, and P_2 holds $(m = x - r, r_2)$ where $r = r_1 + r_2$.

As one of our main results, we propose a 2-round secure comparison protocol Π_{SignBit} in the semi-honest setting. Note that in PPML over $\mathbb{Z}_{2^\ell}$, the secure comparison problem is equivalent to the sign bit extraction problem, i.e. checking if $\mathsf{sign}(x) = 0$ where $\mathsf{sign}(x)$ denotes the sign bit (a.k.a., the left-most bit) of x. Intuitively, our protocol works as follows. For $a \in \mathbb{Z}_{2^\ell}$, let $\hat{a} := a - 2^{\ell-1} \cdot \mathsf{sign}(a)$ denote the value a after removing its sign bit. Hence, $m := \mathsf{sign}(m) \| \hat{m}$ and $r := \mathsf{sign}(r) \| \hat{r}$. Observe that since $x = m + r \bmod 2^\ell$, the sign bit of x equals to $\mathsf{sign}(r) \oplus \mathsf{sign}(m) \oplus (\hat{m} \overset{?}{\geq} 2^{\ell-1} - \hat{r})$, where the boolean check $(\hat{m} \overset{?}{\geq} 2^{\ell-1} - \hat{r})$ represents the carry bit from $\hat{m} + \hat{r}$. Therefore, our main task is to obliviously evaluate $(\hat{m} \overset{?}{\geq} 2^{\ell-1} - \hat{r})$, where $2^{\ell-1} - \hat{r}$ held by P_0 and $\hat{m}$ held by both P_1 and P_2. For this comparison, we can locate and check the highest different bit of $2^{\ell-1} - \hat{r}$ and $\hat{m}$ in binary. Let $s := \hat{m} \oplus (2^{\ell-1} - \hat{r})$. Notice that the position of the highest different bit between $2^{\ell-1} - \hat{r}$ and $\hat{m}$ is equivalent to the position of the first non-zero bit of s. Denote such a position as ζ, and denote the ζ-th bit of $\hat{m}$ as $\hat{m}_\zeta$. It is easy to see that $\hat{m}_\zeta = (\hat{m} \overset{?}{\geq} 2^{\ell-1} - \hat{r})$.

Without considering security, $\hat{m}_\zeta$ can be determined through the following steps. (i) Compute s' as the prefix-sum of s, i.e., $s'_i := \sum_{k=0}^{i} s_k$ for $i \in \mathbb{Z}_\ell$. (ii) Compute $s''_i := s'_i - 2s_i + 1$. We argue that s'' will only contain one zero at the position of the first non-zero bit of s. Indeed, it converts all the prefix zero bits of s' to 1 (namely, if $s'_i = 0 \wedge s_i = 0$ then $s''_i = 1$); it converts the first non-zero bit of s' to 0 (namely, if $s'_i = 1 \wedge s_i = 1$ then $s''_i = 0$); it converts the suffix bits to non-zero values (namely, in case $s_i = 0$, $s'_i \geq 1$, we have $s''_i = s'_i - 2s_i + 1 \geq 2$; in case $s_i = 1$, $s''_i \geq 2$, we have $s''_i = s'_i - 2s_i + 1 \geq 1$). (iii) P_1 and P_2 sends $\hat{m}$ and s'' to P_0; P_0 then locates ζ as the position of the only zero bit in s'', and outputs $\hat{m}_\zeta$ as the comparison result.

Achieving malicious security in the one-bit leakage model [23], [26]. To minimize the overhead while converting the above semi-honest protocol to withstand a malicious adversary, we introduce the batch verification technique. More specifically, we design a verification protocol Π_{VSignBit} to check the correctness of multiple semi-honest secure comparison protocols at the same time. Our main observation is that if we introduce an IT-secure MAC (Cf. TABLE. 2, below) to the share of s'' on top of the semi-honest version, P_0 can verify the correctness of s'' through the MAC check, which prevents malicious P_1 or P_2 from tampering with s''. Next, since there is at most one malicious adversary among the 3 parties under static corruption, we can adopt the dual execution paradigm [26] and perform the verification protocol twice, but switch the role of the players, i.e., we nominate a different party to play the role of the P_0 and let him generate an IT-secure MAC and check the execution correctness. The comparison result shall be accepted if and only if both verifications pass (Cf. Sec. 4.2 for details).

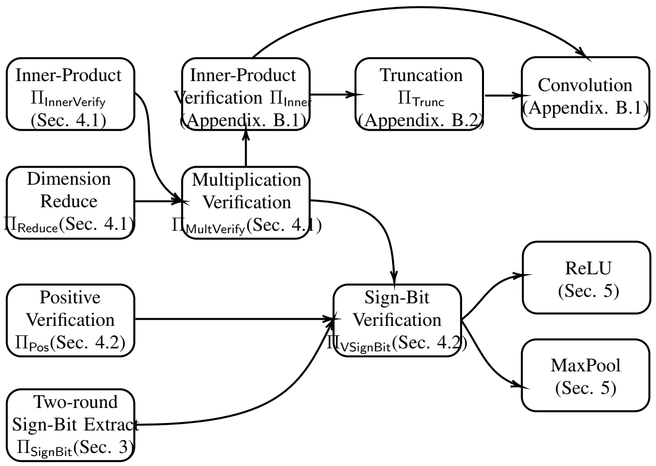


Figure 1: The roadmap of Aegis

Analogously, for the malicious multiplication, the parties first invoke the semi-honest multiplication protocol, and perform a batch verification at the end. Goyal *et al.* [22] proposes a technique that can transfer the verification of N dimension inner product triple to the verification of $N/2$ dimension inner product with constant overhead. However, Goyal *et al.* [22] works on Shamir's secret sharing, which is performed over a prime-order field, naively converting their protocol to the ring setting could cause the soundness issue. Also as mentioned above, the techniques [14], [18], [24] to adopt the multiplication verification over the field to the ring is not suitable for the protocol proposed in [22]. To resolve the soundness issue, we extend the shared elements over $\mathbb{Z}_{2^\ell}$ to the quotient ring of polynomials $\mathbb{Z}_{2^\ell}[x]/f(x)$ [4], [6], [7], where $f(x)$ is a degree-d irreducible polynomial over $\mathbb{Z}_{2^\ell}$ in order to apply the Lagrange interpolating based dimension reduction technique [22] (Cf. Sec. 4.1). Consequently, the overall communication of our batch multiplication verification protocol is logarithmic to the number of multiplication gates.

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 6 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

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 computational security parameter of GC, and $\lambda = 6$ is the statistical security parameter.)

Operation	Protocol	Offline	Online		Malicious
		Communication (bits)	Rounds	Communication (bits)	
Mult	ABY3 [30]	12ℓ	1	9ℓ	✓
	BLAZE [33]	3ℓ	1	3ℓ	✓
	SWIFT [27]	3ℓ	1	3ℓ	✓
	Ours	1ℓ	1	2ℓ	✓
Inner Product	ABY3 [30]	$12n\ell$	1	$9n\ell$	✓
	BLAZE [33]	$3n\ell$	1	3ℓ	✓
	SWIFT [27]	3ℓ	1	3ℓ	✓
	Ours	1ℓ	1	2ℓ	✓
Inner Product with Trunction	ABY3 [30]	$12n\ell + 84\ell$	1	$9n\ell + 3\ell$	✓
	BLAZE [33]	$3n\ell + 2\ell$	1	3ℓ	✓
	SWIFT [27]	15ℓ	1	3ℓ	✓
	Ours	7ℓ	1	2ℓ	✓
DReLU	ABY3 [30]	60ℓ	$3 + \log \ell$	45ℓ	✓
	BLAZE [33]	$5\kappa\ell + 6\ell + \kappa$	4	$\kappa\ell + 6\ell$	✓
	SWIFT [27]	21ℓ	$3 + \log \ell$	16ℓ	✓
	Falcon [38]	$-$	$5 + \log \ell$	32ℓ	✓
	Bicoptor [43]	0	2	$(\ell^* + \ell)(2 + \ell)$	×
	Ours (Semi-honest)	$(\ell - 1)\log \ell + 2\ell$	2	$4\ell(\log \ell + 1) + 2\ell$	×
	Ours (Malicious)	$(\ell - 1)\log \ell + 2\ell$	2	$2((\lambda + 1)(\ell - 1)\log \ell + 6\ell \log \ell + \ell)$	✓

to construct the ReLU and MaxPool protocols. Compared with CrypTFlow [28] (8-round with $6\ell log\ell + 14\ell$ bits communication) and Bicoptor [43] (2-round with the $(\ell^* + \ell)(2 + \ell)$ bits communication, with error probability $2^{1-\ell^*}$), our protocol demonstrates significant improvements (2-round with $4\ell \log \ell + 2\ell$ 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× speedup over SOTA.

Sign bit verification with Malicious Security. To achieve maliciously secure sign bit extraction, we adopt SPDZ style IT-secure MAC [17] and dual execution technique [26]. The resulting protocol only requires a 2-round with $2((\lambda + 1)(\ell - 1)\log \ell + 6\ell \log \ell)$ 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 [33] (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 the round complexity by 50% and the communication by 75%, 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 Garbled Circuit. In real-world benchmark tests, our protocol exhibits 100× speedup over the Garbled Circuit solution [33] and 6× speedup over the logarithmic rounds solution [30].

Batch verification for multiplication over the 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. 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 proposed by Dalskov *et al.* [16], our protocol reduces the overall communication by 40%. Note that Dalskov *et al.* [16] 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 [27], 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 50% of that in SWIFT [27].

Paper Organization. As shown in Fig. 1, we first propose semi-honest secure sign-bit extraction protocol Π_{SignBit} in Sec. 3. In Sec. 4, we propose our maliciously secure protocols. In Sec. 4.1, we 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. Combining $\Pi_{\mathsf{InnerVerify}}$ and Π_{Reduce}, we obtain the batch multiplication verification protocol $\Pi_{\mathsf{MultVerify}}$, which can verify multiple multiplication triples at once. In Sec. 4.2, 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. Combining Π_{Pos} with $\Pi_{\mathsf{MultVerify}}$, we construct the batch sign bit verification protocol Π_{VSignBit}, which can verify multiple sign bit extraction pairs in at once. In Sec. 5, we build the ReLU protocol Π_{ReLU} and the MaxPool protocol by integrating the above basic protocols. In Appendix. B, we construct other components for machine learning, such as convolution and truncation. In Sec. 6, we benchmark the performance of our protocols.

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 [30], [33]: 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. We use subscripts x_i to represent elements in a vector. When we process each bit of the ring element x, we abuse the representation of subscripts x_i to denote the i^{th} bit from big-endian. We denote $\gamma(x) = \alpha \cdot x$ as the MAC of x where α is the MAC key. We take λ numbers of MAC keys for soundness. We denote $\mathsf{sign}(x)$ as the sign bit of x. We take κ as the security parameter. We use $\eta_{j,k}$ to denote the common seed held by P_j and P_k. 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_j for $j \in \{1, 2\}$ hold share $[x]_j$.
- $\langle\cdot\rangle$-sharing: We define $\langle\cdot\rangle$-sharing over ring $\mathbb{Z}_{2^\ell}$ as $\langle x \rangle := ([r_x], m_x)$ where r_x is a fresh random value and $m_x = r_x + x$. P_j for $j \in \{1, 2\}$ hold $(m_x \in \mathbb{Z}_{2^\ell}, [r_x]_j \in \mathbb{Z}_{2^\ell})$ and P_0 holds $([r_x]_1, [r_x]_2)$.
- $\llbracket\cdot\rrbracket^{p,k}$-sharing: We define $\llbracket\cdot\rrbracket^{p,k}$ over finite field $\mathbb{Z}_p$ as $\llbracket x \rrbracket^p := (\llbracket x \rrbracket_{k+1} \in \mathbb{Z}_p, \llbracket x \rrbracket_{k-1} \in \mathbb{Z}_p)$ where $x = \llbracket x \rrbracket_{k+1} + \llbracket x \rrbracket_{k-1} \pmod p$. P_j for $j \in \{k+1, k-1\}$ hold share $\llbracket x \rrbracket_j$.
- $\|\cdot\|^{p,\lambda,k}$-sharing: We define $\|\cdot\|^{p,\lambda,k}$-sharing over finite field $\mathbb{Z}_p$ as $\|x\|^{p,\lambda,k} := (\llbracket x \rrbracket^p, \{\llbracket \alpha_j \rrbracket^p, \llbracket \gamma(x)_j \rrbracket^p\}_{j \in \mathbb{Z}_\lambda})$. In our sign-bit verification protocol, one party P_k holds $\{\alpha_j\}_{j \in \mathbb{Z}_\lambda}$ which are the plaintext of MAC keys, and the other parties P_{k-1} and P_{k+1} hold the share $(\llbracket x \rrbracket_i, \{\llbracket \alpha_j \rrbracket_i, \llbracket \gamma(x)_j \rrbracket_i\}_{j \in \mathbb{Z}_\lambda})$ for $i \in \{k-1, k+1\}$.

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 $\|\cdot\|^{p,\lambda,k}$ we utilize superscript k to denote that the MAC keys are held by P_k. Note that we let any two shared values $\|x\|^{p,\lambda,k}$ and $\|y\|^{p,\lambda,k}$ for the same key's holder P_k use the same MAC key. For simplicity, we use $\|\cdot\|$, $\llbracket\cdot\rrbracket$ 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 $\langle\cdot\rangle$, $\llbracket\cdot\rrbracket$, and $[\cdot]^{\mathbb{Z}_{2^\ell}[x]}, \langle\cdot\rangle^{\mathbb{Z}_{2^\ell}[x]}$. For $\|\cdot\|$, we have $\|x\| + \|y\| = (\llbracket x \rrbracket + \llbracket y \rrbracket, \{\llbracket \alpha_j \rrbracket, \llbracket \gamma(x)_j \rrbracket + \llbracket \gamma(y)_j \rrbracket\}_{j \in \mathbb{Z}_\lambda})$, $c \cdot \|x\| = (c \cdot \llbracket x \rrbracket, \{\llbracket \alpha_j \rrbracket, c \cdot \llbracket \gamma(x)_j \rrbracket\}_{j \in \mathbb{Z}_\lambda})$ and $c + \|x\| = (c + \llbracket x \rrbracket, \{\llbracket \alpha_j \rrbracket, c \cdot \llbracket \alpha_j \rrbracket + \llbracket \gamma(x)_j \rrbracket\}_{j \in \mathbb{Z}_\lambda})$.

Secret sharing. Let $\Pi_{[\cdot]}$, $\Pi_{\llbracket\cdot\rrbracket}$, $\Pi_{\langle\cdot\rangle}$, 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 [42]. In our protocol description, when we let parties P_j and P_k pick random values together, we mean that these parties invoke PRG with seed $\eta_{j,k}$. The brief sketch of secret sharing schemes is as follows.

- $[x] \leftarrow \Pi^\ell_{[\cdot]}(x)$: (Generate shares of x.)
  - P_0 and P_1 pick random value $[x]_1 \in \mathbb{Z}_{2^\ell}$ with seed $\eta_{0,1}$;
  - P_0 sends $x_2 = x - [x]_1 \pmod{2^\ell}$ to P_2.
- $[x] \leftarrow \Pi^\ell_{[\cdot]}$: (Generate shares of a random value.)
  - P_0 and P_1 pick random value $[x]_1 \in \mathbb{Z}_{2^\ell}$ with seed $\eta_{0,1}$;
  - P_0 and P_2 pick random value $[x]_2 \in \mathbb{Z}_{2^\ell}$ with seed $\eta_{0,2}$;
  - P_0 calculates $x = [x]_1 + [x]_2$.
- $\llbracket x \rrbracket \leftarrow \Pi^{p,k}_{\llbracket\cdot\rrbracket}(x)$: (Generate shares of x.)
  - P_k and P_{k+1} pick random value $\llbracket x \rrbracket_{k+1} \in \mathbb{Z}_p$ with seed $\eta_{k,k+1}$;
  - P_k sends $\llbracket x \rrbracket_{k-1} = x - \llbracket x \rrbracket_{k+1} \pmod p$ to P_{k-1}.
- $\llbracket x \rrbracket \leftarrow \Pi^{p,k}_{\llbracket\cdot\rrbracket}$: (Generate shares of a random value.)
  - P_k and P_{k+1} pick random value $\llbracket x \rrbracket_{k+1} \in \mathbb{F}_p$ with seed $\eta_{k,k+1}$;
  - P_k and P_{k-1} pick random value $\llbracket x \rrbracket_{k-1} \in \mathbb{F}_p$ with seed $\eta_{k-1,k}$;
  - P_k calculates $x = \llbracket x \rrbracket_{k+1} + \llbracket x \rrbracket^p_{k-1}$.
- $\langle x \rangle \leftarrow \Pi^{\ell,k}_{\langle\cdot\rangle}(x)$: (Generate shares of x.)
  - All parties perform $[r_x] \leftarrow \Pi_{[\cdot]}$ in the offline phase, and P_k holds both seeds of $[r_x]_1$ and $[r_x]_2$ generation;
  - P_i send $m_x = x + [r_x]_1 + [r_x]_2$ to P_1 and P_2.
- $\langle x \rangle \leftarrow \Pi^\ell_{\langle\cdot\rangle}$: (Generate shares of a random value.)
  - All parties perform $[r_x] \leftarrow \Pi_{[\cdot]}$ in the offline phase;
  - P_1 and P_2 pick random value m_x with seed $\eta_{1,2}$.
- $\|x\| \leftarrow \Pi^{p,\lambda,k}_{\|\cdot\|}(x)$: (Generate shares of x.)
  - All parties invoke $\llbracket \alpha_j \rrbracket \leftarrow \Pi^{p,k}_{\llbracket\cdot\rrbracket}$ for $j \in \mathbb{Z}_\lambda$;
  - P_k calculates $\gamma(x)_j = x \cdot \alpha_j$ for $j \in \mathbb{Z}_\lambda$;
  - All parties invoke $\llbracket \gamma(x)_j \rrbracket \leftarrow \Pi^{p,k}_{\llbracket\cdot\rrbracket}(\gamma(x)_j)$ for $j \in \mathbb{Z}_\lambda$ and $\llbracket x \rrbracket \leftarrow \Pi^{p,k}_{\llbracket\cdot\rrbracket}(x)$.

$\Pi_{[\cdot]}$ 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 $\Pi^{\ell[x]}_{[\cdot]}$, $\Pi^{\ell[x]}_{\langle\cdot\rangle}$.

Verifiability of share reconstruction. We note that the shared form $\langle\cdot\rangle$ has the verifiable reconstruction 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

TABLE 2: The share structure of Aegis. (For $\llbracket\cdot\rrbracket^{p,k}$ and $\lVert\cdot\rVert^{p,\lambda,k}$, the example in the table depicts the case of $\llbracket\cdot\rrbracket^{p,0}$ and $\lVert\cdot\rVert^{p,\lambda,0}$)

	$\llbracket x\rrbracket^{p,0}$	$\lVert x\rVert^{p,\lambda,0}$	$[x]$	$\langle x\rangle$
P_0	$-$	$\{\alpha_j\}_{j\in\mathbb{Z}_\lambda}$	$-$	$([r_x]_1,[r_x]_2\in\mathbb{Z}_{2^\ell})$
P_1	$\llbracket x\rrbracket_1^p\in\mathbb{Z}_p$	$(\llbracket x\rrbracket_1^p,\{\llbracket\alpha_j\rrbracket_1^p,\llbracket\gamma(x)_j\rrbracket_1^p\}_{j\in\mathbb{Z}_\lambda})$	$[x]_1\in\mathbb{Z}_{2^\ell}$	$([r_x]_1,m_x=r_x+x)$
P_2	$\llbracket x\rrbracket_2^p\in\mathbb{Z}_p$	$(\llbracket x\rrbracket_2^p,\{\llbracket\alpha_j\rrbracket_1^p,\llbracket\gamma(x)_j\rrbracket_2^p\}_{j\in\mathbb{Z}_\lambda})$	$[x]_2\in\mathbb{Z}_{2^\ell}$	$([r_x]_2,m_x=r_x+x)$

shares will be detected by the honest parties. In addition, the shared form $\|\cdot\|^{p,k}$ also maintains the verifiability when one of the P_{k-1}, P_{k+1} is malicious. Because P_k can assert the correctness of share through the MAC check. We apply the hash function H to reduce the communication of $\|x\|$ reconstruction [15], where the duplicated messages will be packaged into the single hash message. Formally, the verifiable reconstruct 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 $H([r_x]_1)$ to P_2;
  - P_2 sends $H(m_x)$ to P_0 and $H([r_x]_2)$ to P_1;

  If the received messages from the other parties are inconsistent, P_i output abort. Otherwise P_i output $x=m_x-[r_x]_1-[r_x]_2$.
- $x\leftarrow\Pi_{\mathsf{Rec}}^{\ell,k}(\langle x\rangle)$: All parties send their shares (or the hash value) to P_k. If the received messages from the other parties are inconsistent, P_k output abort. Otherwise P_k output $x=m_x-[r_x]_1-[r_x]_2$.
- $x\leftarrow\Pi_{\mathsf{Rec}}^{p,k}(\|x\|)$:
  - Each party P_i for $i\neq k$ sends its shares $\llbracket x\rrbracket_i,\{\llbracket\gamma(x)_j\rrbracket_i\}_{j\in\mathbb{Z}_\lambda}$ to P_k;
  - P_k 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, $\Pi_{\mathsf{Rec}}^{\ell[x]}$ works analogously as the above.

Preprocessing and postprocessing. We follow the "preprocessing" paradigm [3] 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 [24] where batch verification is performed.

Multiplication gate. We adopt the multiplication protocol of ASTRA [10]. 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

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

$[\Gamma']=m_xm_y-m_x[r_y]-m_y[r_x]$ 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 $F^d(x_1,\dots,x_n)=y$, we design a evaluation protocol $\langle y\rangle=\Pi_{\mathsf{PolyEvl}}(F^d,\langle x_1\rangle,\dots,\langle x_n\rangle)$ following the design of multiplication gate. In particular, plugin the underlying shares, we have

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

Let $\mathcal{I}_k$ be the k^{th} term of $F^d(x_1,\dots,x_n)=\sum_{k=0}^{m}c_k\cdot\prod_{x_{s_j}\in\mathcal{I}_k}x_{s_j}$. 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, whereas the online communication is still 2ℓ to reconstruct m_y. Let $\Pi_{\mathsf{PolyEvl}}^{\ell[x]}$ 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 [12], a 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.

Security Model. We analyze the security of our protocols in the well-known Universal Composibility (UC) framework [9], 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 $\mathcal{F}$ with the excuted function f. All parties send their share to $\mathcal{F}$, $\mathcal{F}$ calculate and output the result depending on the adversary $\mathcal{S}$.

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:

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

3. Secure Sign Bit Extraction

In this section, we propose a novel sign bit extraction protocol Π_{SignBit}. For sign bit extraction function $z = \mathsf{sign}(x)$, protocol Π_{SignBit} can output $\langle z \rangle$ from input $\langle x \rangle$. In Sec. 4, we apply it to the malicious setting.

3.1. Intuition

We aim to design a two-round sign bit extraction protocol. Intuitively, we want the protocol to look like this: Firstly, P_1 and P_2 perform some local transform to produce some shared material for a predicate which implies the sign bit extraction result. In the first communication round, P_1 and P_2 reveal the material to P_0, in the second round, P_0 performs the predicate check and reshares the result to P_1 and P_2. In what follows, we specifically analyze how to construct such a predicate without losing privacy. Considering $\langle x \rangle := \{m_x, [r_x]\}$ and $x = m_x + (-r_x)$, the sign bit of x can be obtained by two parts of XOR operations. One part is the XOR of the sign bits of m_x and $-r_x$, and the other part is the carry from the sum of the low bits (excluding the sign bit) of m_x and $-r_x$. Namely, $\mathsf{sign}(x) := (\hat{m_x} + \hat{r_x} \overset{?}{\geq} 2^{\ell-1}) \oplus \mathsf{sign}(-r_x) \oplus \mathsf{sign}(m_x)$, where we denote $m_x := \mathsf{sign}(m_x)||\hat{m_x}$ and $-r_x := \mathsf{sign}(-r_x)||\hat{r_x}$. Among them, $\mathsf{sign}(-r_x)$ and $\mathsf{sign}(m_x)$ can be evaluted locally. For the remaining part $\hat{m_x} + \hat{r_x} \overset{?}{\geq} 2^{\ell-1}$, we observe that it is equivalent to the boolean check of $\hat{m_x} \overset{?}{\geq} 2^{\ell-1} - \hat{r_x}$ (It works due to $2^{\ell-1} \geq \hat{r_x}$), which is millionaire problem while P_1 and P_2 hold $\hat{m_x}$, P_0 holds $2^{\ell-1} - \hat{r_x}$. For the convenience of presentation, we will use a and b to denote $\hat{m_x}$ and $2^{\ell-1} - \hat{r_x}$ below. We solve this millionaire problem as follows.

First non-zero bit position detection problem. We first convert the millionaire problem $a \overset{?}{\geq} b$ to the first non-zero position detection problem: $a \overset{?}{\geq} b$ equal to a_ζ for the $\zeta \in \mathbb{Z}_\ell$ which is the first non-zero position of list $\mathcal{L}_1 := \{m_i\}_{i \in \mathbb{Z}_\ell}$. We explain how the conversion works as follows. When we view a and b as XOR shares, namely, $m = a \oplus b$, the first non-zero bit of m (denoted its index as ζ) represents the highest different bit of a and b whose corresponding position ζ can be used to determine the comparison result, that is, $a_\zeta = a \overset{?}{\geq} b$. (Note that if $a = b$, there is no non-zero bit in m; therefore, we append 1 to a and 0 to b, ensuring $a \neq b$.) Next, we apply a transform to convert finding the first non-zero bit problem to finding the position of the only zero element in a list.

First non-zero bit extraction transform. Let $\mathcal{L}_1 := \{m_i\}_{i \in \mathbb{Z}_\ell}$ be the list of the individual bits of the value $m = a \oplus b$. If we calculate its prefix sum $m'_i = \sum_{t=0}^{i} m_t$, it is easy to see that all prefixes are zero until the first non-zero bit. Then we calculate $m''_i = m'_i - 2m_i + 1$, which converts all the prefix zero bits to 1 (that is, if $m_i = 0, m'_i = 0$ then $m''_i = 1$), converts the first non-zero bit to zero (that is, if $m_i = 1, m'_i = 1$ then $m''_i = 0$) and converts the suffix bits to non-zero value (that is, in case $m_i = 0$, $m'_i \geq 1$, we have $m''_i = m'_i - 2m_i + 1 \geq 2$; in case $m_i = 1$, $m''_i \geq 2$, we have $m''_i = m'_i - 2m_i + 1 \geq 1$). Therefore, m'' will only contain one zero at the position of the first non-zero bit of m. In addition, considering $m'_i - 2m_i + 1 \leq \ell + 1$, in order to avoid unexpected zero triggered by wrapping round, all the calculations should be performed on $\mathbb{Z}_p$ where prime $p > \ell + 1$. Formally, we define this transform as $\mathcal{L}_2 = \phi(\mathcal{L}_1) := \{\sum_{t=0}^{i} m_t - 2 \cdot m_i + 1 \bmod p\}_{i \in \mathbb{Z}_\ell}$. We have Theorem 1, and its proof can be found in Appendix. A.1.

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

P_j and P_k hold the common seed $\eta_{j,k} \in \{0,1\}^{\lambda}$.
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 $\eta_{1,2}$ and reveals $[\Gamma] = \Delta + [r'] - 2\Delta \cdot [r'] + [r_z]$ to each other.
- P_0 does:
  1) calculate $\hat{r}_x = -r_x - \mathsf{sign}(-r_x) \cdot 2^{\ell-1} \in \mathbb{Z}_{2^{\ell-1}}$
  2) extract $2^{\ell-1} - \hat{r}_x$ as $\{r_{x,0}, \ldots, r_{x,\ell-2}\}$
  3) perform $[\![r_{x,i}]\!]^p \leftarrow \Pi^p_{[\![\cdot]\!]}(r_{x,i})$ for $i \in \mathbb{Z}_{\ell-1}$, taking the biggest prime of $p \in (\ell, 2^{\log \ell + 1}]$;

Online:

- P_j, for $j \in \{1,2\}$ does:
  1) set $\hat{m}_x = m_x - \mathsf{sign}(m_x) \cdot 2^{\ell-1}$ and bitexact it as $\{\hat{m}_{x,i} \in \{0,1\}\}_{i \in \mathbb{Z}_{\ell-1}}$ while $\sum_{i=0}^{\ell-2} 2^{\ell-2-i} \hat{m}_{x,i} = \hat{m}_x$;
  2) set $\hat{m}_{x,\ell-1} = 1$ and $[\![r_{x,\ell-1}]\!] = [\![0]\!]$;
  3) set $[\![m_i]\!]^p = \hat{m}_{x,i} + [\![r_{x,i}]\!]^p - 2\hat{m}_{x,i} \cdot [\![r_{x,i}]\!]^p$ for $i \in \mathbb{Z}_\ell$.
  4) pick same random values $\{w_i, w'_i\}_{i \in \mathbb{Z}_\ell} \in (\mathbb{Z}_p^*)^{2\ell}$ via PRF with seed $\eta_{1,2}$;
  5) calculate $[\![m'_i]\!]^p = \sum_{t=1}^{i} [\![m_t]\!]^p - 2 \cdot [\![m_i]\!]^p + 1$ and $[\![u_i]\!]^p = w_i \cdot [\![m'_i]\!]^p + (\mathsf{sign}(m_x) \oplus \hat{m}_{x,i} \oplus \Delta)$ and $[\![u'_i]\!]^p = w'_i(w_i \cdot [\![m'_i]\!]^p + 1)$ for $i \in \mathbb{Z}_\ell$;
  6) pick a random permutation π via PRF with seed $\eta_{1,2}$ and permute the list $\{[\![\hat{u}_i]\!]^p\}_{i \in \mathbb{Z}_\ell} = \pi(\{[\![u_i]\!]^p\}_{i \in \mathbb{Z}_\ell})$ and $\{[\![\hat{u}'_i]\!]^p\}_{i \in \mathbb{Z}_\ell} = \pi(\{[\![u'_i]\!]^p\}_{i \in \mathbb{Z}_\ell})$;
  7) reveal $\{[\![\hat{u}_i]\!]^p\}_{i \in \mathbb{Z}_\ell}$ and $\{[\![\hat{u}'_i]\!]^p\}_{i \in \mathbb{Z}_\ell}$ to P_0;
- P_0 sends $m' = \mathsf{sign}(-r_x) - r'$ if $\exists \hat{u}_i = 0 \wedge \hat{u'_i} \neq 0$ for $i \in \mathbb{Z}_\ell$ else $m' = (1 \oplus \mathsf{sign}(-r_x)) - r'$ to P_j, for $j \in \{1,2\}$;
- P_j, for $j \in \{1,2\}$ sets $m_z = m' - 2\Delta \cdot m' + \Gamma$;
- All parties output $\langle z \rangle = ([r_z], m_z)$.

Figure 2: The Sign Bit Extraction Protocol.

Theorem 1. *Let $\mathcal{L} := (L_0, \ldots, L_{\ell-1}) \in \{0,1\}^{\ell}$ be a binary vector. There exists a linear transformation ϕ such that*

Functionality $\mathcal{F}_{\mathsf{SignBit}}[\mathbb{Z}_{2^\ell}]$

$\mathcal{F}_{\mathsf{SignBit}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$.

Input:

- Upon receiving $(\mathsf{Input}, \mathsf{sid}, (r_1, r_2))$ from P_0, send $(\mathsf{Input}, \mathsf{sid}, P_0)$ to $\mathcal{S}$ and record $(r_1, r_2) \in (\mathbb{Z}_{2^\ell})^2$;
- Upon receiving $(\mathsf{Input}, \mathsf{sid}, (m_j, r'_j))$ from P_j, $j \in \mathbb{Z}_2$, send $(\mathsf{Input}, \mathsf{sid}, P_j)$ to $\mathcal{S}$ and record $(m_j, r'_j) \in (\mathbb{Z}_{2^\ell})^2$;

Execution:

- If $m_1 = m_2$, compute $z := \mathsf{sign}(m_1 - r_1 - r_2)$;
- If $m_1 \neq m_2$, send $(\mathsf{Error}, \mathsf{sid})$ to $\mathcal{S}$; upon receiving $(\mathsf{Compute}, \mathsf{sid}, \mathsf{Alg})$ from $\mathcal{S}$, compute $z := \mathsf{Alg}(r, m_1, m_2)$;
- Pick random $u_1, u_2 \leftarrow \mathbb{Z}_{2^\ell}$, set $u := u_1 + u_2$ and $w := z + u$;
- Upon receiving $(\mathsf{Modify}, \mathsf{sid}, \{\delta_i\}_{i\in\mathbb{Z}_6})$, send $(\mathsf{Output}, \mathsf{sid}, (u_1 + \delta_0, u_2 + \delta_1))$ to P_0, $(\mathsf{Output}, \mathsf{sid}, (w + \delta_2, u_1 + \delta_3))$ to P_1, $(\mathsf{Output}, \mathsf{sid}, (w + \delta_4, u_2 + \delta_5))$ to P_2.

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

$\phi(\mathcal{L}) = (L'_0, \dots, L'_{\ell-1})$ satisfies:

- *Let $i^* \in \mathbb{Z}_\ell$ be the index of the first non-zero bit in $\mathcal{L}$, that is, $L_{i^*} = 1 \ \wedge \ \forall i < i^* : L_i = 0$.*
- *$L'_{i^*} = 0$ and $L'_j \neq 0$ for all $j \neq i^*$.*

Thanks to the replicated share structure, we can easily convert each XOR shared bit m_i (that is, $m_i = a_i \oplus b_i$) to $[\![m_i]\!]^p$. We let P_0 secret shares $[\![b_i]\!]^p$ to P_1 and P_2 so that P_1 and P_2 can calculate $[\![m_i]\!]^p = a_i + [\![b_i]\!]^p - 2 \cdot a \cdot [\![b_i]\!]^p$ locally.

Oblivious bit detection. So far, we still cannot directly reveal the list $\mathcal{L}_2 := \{s_i \in \mathbb{Z}_p\}_{i\in\mathbb{Z}_\ell}$ and a to P_0 for checking a_ζ with the zero value position $\zeta \in \mathbb{Z}_\ell$, due to the privacy concerns. We make P_0 detect a_ζ obliviously as follows. Since the element $\mathcal{L}_2$ is over field $\mathbb{Z}_p$, we can scale each element with random value $w_i \in \mathbb{Z}_p^*$ which masks each value except zero. Subsequently, we store each a_i in the s_i by adding a_i to the masked value, namely, $u_i = a_i + w_i \cdot s_i \pmod p$. This storage ensures the following property: the 0 value of list $\{u_i\}_{i\in\mathbb{Z}_\ell}$ is obtained by one of (i) $a_\zeta = 0 \wedge s_\zeta = 0$, (ii) $a_i = 1 \wedge w_i \cdot s_i = p - 1$. When we exclude all of the cases of (ii), we can obtain a_ζ though checking the existence of zero elements in list $\{u_i\}_{i\in\mathbb{Z}_\ell}$. To exclude the second case, we introduce the second random mask value $w'_i \in \mathbb{Z}_p^*$, and calculate $u'_i = w'_i \cdot (w_i \cdot s_i + 1)$ for $i \in \mathbb{Z}_\ell$. The zero item of list $\{u'_i\}_{i\in\mathbb{Z}_\ell}$ implicit that whether $w_i \cdot s_i = p - 1$. Put all together, we get the new predicate: there exists $i \in \mathbb{Z}_\ell$ such that $u_i = 0 \wedge u'_i \neq 0$ for the list $\{u_i\}_{i\in\mathbb{Z}_\ell}$ and $\{u'_i\}_{i\in\mathbb{Z}_\ell}$. To further protect privacy, we employ same random permutation π on $\{u_i\}_{i\in\mathbb{Z}_\ell}$ and $\{u'_i\}_{i\in\mathbb{Z}_\ell}$ which does not affect the predicate relationship.

Protect a_ζ with mask. Directly reveal $\{u_i\}_{i\in\mathbb{Z}_\ell}$ and $\{u'_i\}_{i\in\mathbb{Z}_\ell}$ to P_0 will leak the comparison result to P_0. We introduce $\Delta \in \{0, 1\}$ which is known to P_1 and P_2 to avoid this leakage. When calculate $\{u_i\}_{i\in\mathbb{Z}_\ell}$, P_1 and P_2 input $\Delta \oplus a_i$ instead of a_i. Finally, revealing $\{u_i\}_{i\in\mathbb{Z}_\ell}$ and $\{u'_i\}_{i\in\mathbb{Z}_\ell}$ to P_0, P_0 can calculate $a_\zeta \oplus \Delta$ through verifying the predicate $(\exists u_i = 0 \wedge u'_i \neq 0, \forall \{u_i\}_{i\in\mathbb{Z}_\ell}, \forall \{u'_i\}_{i\in\mathbb{Z}_\ell})$.

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.

The second round: reshre $\langle a_\zeta\rangle$. Considering the result of the first round, P_0 holds the predicate check result $a_\zeta \oplus \Delta$, P_1 and P_2 hold the mask value Δ. We observe that it can be transferred to $\langle a_\zeta\rangle$ in a single round with 2ℓ bits communication. We asume that $z = a_\zeta$ and $\langle z\rangle := \{m_z, [r_z]\}$. We first let all parties locally generate random share $[r']$ and $[r_z]$, where P_0 holds r', P_1 and P_2 hold two-party share $[r']$. Then P_1 and P_2 calculate $[\Gamma] = \Delta + [r'] - 2\Delta \cdot [r'] + [r_z]$ and reveal to each other in the offline phase. After getting $z \oplus \Delta$, P_0 send $z \oplus \Delta - r' \in \mathbb{Z}_{2^\ell}$ to both P_1 and P_2. It is easy to see that $(1 - 2\Delta)(z \oplus \Delta - r') + \Gamma = r_z + z = m_z$, where P_1 and P_2 hold Δ and Γ so that they can calculate m_z locally.

3.2. Concrete Construction

By filling in some detailed descriptions, we complete our protocol, which is depicted in Fig. 2. Next, we will explain our protocol step by step as follows.

- In the offline phase, P_1 and P_2 generate Δ to mask the sign bit and Γ for the second round resharing. P_0 split the sign bit of $-r_x$ and the remain part $\hat{r}_x$. As mentioned before, the sign bit $\mathsf{sign}(x)$ equal to $(\hat{m}_x + \hat{r}_x \geq 2^{\ell-1}) \oplus \mathsf{sign}(-r_x) \oplus \mathsf{sign}(m_x)$. P_0 bit-extract $2^{\ell-1} - \hat{r}_x$ for the comparison $\hat{m}_x + \hat{r}_x \geq 2^{\ell-1}$, and share each bit in the field $\mathbb{Z}_p$.
- In steps 1-3, P_1 and P_2 set $[\![m_i]\!]^p$, where m_i represents the i-th bit of $\hat{m}_x \oplus (2^{\ell-1} - \hat{r}_x)$. The transformation can be locally performed. Moreover, we set $\hat{m}_{x,\ell} = 1$ and $[\![r_{x,\ell}]\!] = [\![0]\!]$ to ensure that protocol output equals to 1 when $\hat{m}_x + \hat{r}_x = 2^{\ell-1}$.
- In step 5, P_1, P_2 transfer $[\![m_i]\!]^p$ to $[\![m'_i]\!]^p$ via the transformation ϕ and generate the aforementioned lists $\{u_i\}_{i\in\mathbb{Z}_\ell}$ and $\{u'_i\}_{i\in\mathbb{Z}_\ell}$. Considering $(\hat{m}_x + \hat{r}_x \geq 2^{\ell-1}) \oplus \mathsf{sign}(-r_x) \oplus \mathsf{sign}(m_x)$, we let P_1 and P_2 further XOR the sign bit of m_x, such that P_0 will output $\mathsf{sign}(m_x) \oplus \hat{m}_{x,\zeta} \oplus \Delta$ rather than $\hat{m}_{x,\zeta} \oplus \Delta$.

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
  - $\langle f_i(0)\rangle^{\ell[x]} = \langle x^{(2\cdot i)}\rangle^{\ell[x]}$;$\langle f_i(1)\rangle^{\ell[x]} = \langle x^{(2\cdot i+1)}\rangle$; $\langle f_i(2)\rangle^{\ell[x]} = 2\cdot\langle f_i(1)\rangle^{\ell[x]} - \langle f_i(0)\rangle^{\ell[x]}$;
  - $\langle g_i(0)\rangle^{\ell[x]} = \langle y^{(2\cdot i)}\rangle^{\ell[x]}$;$\langle g_i(1)\rangle^{\ell[x]} = \langle y^{(2\cdot i+1)}\rangle^{\ell[x]}$; $\langle g_i(2)\rangle^{\ell[x]} = 2\cdot\langle g_i(1)\rangle^{\ell[x]} - \langle g_i(0)\rangle^{\ell[x]}$;
  - $\langle h(0)\rangle^{\ell[x]} = \sum\langle f_i(0)\rangle^{\ell[x]}\cdot\langle g_i(0)\rangle^{\ell[x]}$;$\langle h(1)\rangle^{\ell[x]} = \langle z\rangle^{\ell[x]} - \langle h(0)\rangle^{\ell[x]}$; $\langle h(2)\rangle^{\ell[x]} = \sum\langle f_i(2)\rangle^{\ell[x]}\cdot\langle g_i(2)\rangle^{\ell[x]}$;
- 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

- In step 6, P_1, P_2 random shuffle the list $\{u_i\}_{i\in\mathbb{Z}_\ell}$ and $\{u'_i\}_{i\in\mathbb{Z}_\ell}$ with the same permutation π.
- In step 7, P_1, P_2 open $\{u_i\}_{i\in\mathbb{Z}_\ell}$ and $\{u'_i\}_{i\in\mathbb{Z}_\ell}$ to P_0. P_0 can draw the conclusion based on observations of $\{u_i\}_{i\in\mathbb{Z}_\ell}$ and $\{u'_i\}_{i\in\mathbb{Z}_\ell}$: if there exist i that $u_i = 0 \wedge u'_i \neq 0$, then $\mathsf{sign}(m_x) \oplus \hat{m}_{x,\zeta} \oplus \Delta = 0$, otherwise $\mathsf{sign}(m_x) \oplus \hat{m}_{x,\zeta} \oplus \Delta = 1$.
- For the second round of online phase, P_0 further XOR $\mathsf{sign}(-r_x)$ to get $\mathsf{sign}(-r_x) \oplus \mathsf{sign}(m_x) \oplus \hat{m}_{x,\zeta} \oplus \Delta$ which is the masked value of sign bit, stemming from $\mathsf{sign}(x) = \mathsf{sign}(-r_x) \oplus \mathsf{sign}(m_x) \oplus \hat{m}_{x,\zeta}$. Now, P_1 and P_2 hold Δ. We use the aforementioned reshare technique to transfer the XOR shared value $\{\mathsf{sign}(x) \oplus \Delta, \Delta\}$ to $\langle\cdot\rangle$-shared value, with one round and 2ℓ communication.

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

Security. We analyze the security of our sign-bit extraction protocol in the UC framework. We define the functionality $\mathcal{F}_{\mathsf{SignBit}}$ for our sign-bit extraction in Fig. 3. We show that our protocol can ensure privacy against malicious adversaries, and ensure both correctness and privacy against semi-honest adversaries. For the malicious adversary, our functionality $\mathcal{F}_{\mathsf{SignBit}}$ allows the corrupted P_0 to select arbitrary replicated secret shares r_1 and r_2, even inconsistent with other parties' input. For the corrupted P_1 or P_2, $\mathcal{F}_{\mathsf{SignBit}}$ allows the adversary to select the algorithm to calculate output. For the honest adversary, $\mathcal{F}_{\mathsf{SignBit}}$ allows the simulator to input $\mathsf{Alg}(r_1, r_2, m_1, m_2) := \mathsf{sign}(m_1 - r_1 - r_2)$ which evaluation sign bit extraction correctly.

Theorem 2. *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. 2 UC realizes* $\mathcal{F}_{\mathsf{SignBit}}$ *against malicious PPT adversaries who can statically corrupt up to one party.*

Proof. See Appendix A.2. ☐

4. Achieving Malicious Security

Aegis uses the postprocessing verification procedure to detect any potential malicious behavior. We utilize the batch verification paradigm which performs all verification in a single message. We emphasize the need for verification within a single message, which can resist the selective failure attack. We first present our batch verification protocol for multiplication and then introduce batch verification protocol for sign bit extraction.

4.1. Batch Multiplication Verification

Intuitively, to batch verify N multiplication $\{\langle x^{(i)}\rangle, \langle y^{(i)}\rangle, \langle z^{(i)}\rangle\}_{i\in\mathbb{Z}_N}$, we can turn to verify that 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. 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 first 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 Π_{PolyEvl}) with random share $\langle\alpha\rangle$. Since Π_{PolyEvl} is secure up to additive attack [12], 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 $\mathbb{Z}_2$ [4]. (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. Considering the cost of Π_{PolyEvl}, the above solution still requires $\Theta(N)$ communication for the offline phase. However, we observe that the conversion to the ring extension does not introduce any extra communication. Furthermore, we find that the dimension reduction technique of [22] is compitable with ring extension which can be used to reduce the $\Theta(N)$ communication to $\Theta(\log N)$.

Our batch multiplication verification protocol is as follows.

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

Input : A $N$-dimension $\langle\cdot\rangle^{\ell[x]}$-shared inner product pair.
Output : $z \overset{?}{=} \sum_{i=1}^{N} x^{(i)} \cdot y^{(i)}$.
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 $\Pi_{\mathsf{PolyEvl}}^{\ell[x]}$;
- All parties call $\Delta = \Pi_{\mathsf{Rec}}^{\ell[x]}(\langle\Delta\rangle^{\ell[x]})$;
- All parties output $1$ if $\Delta = 0$, otherwise $0$.
```

Figure 6: The Inner Product Verification Protocol

```
Protocol $\Pi_{\mathsf{MultVerify}}^{R}(\{\langle x^{(i)}\rangle, \langle y^{(i)}\rangle, \langle z^{(i)}\rangle\}_{i\in\mathbb{Z}_N})$

Input : $N$ pairs of $\langle\cdot\rangle$-shared multiplication.
Output : $z^{(i)} \overset{?}{=} x^{(i)} \cdot y^{(i)}$ for all $i \in \mathbb{Z}_N$.
Execution:
- All parties invoke $\Pi_{\mathsf{Trans}}(\{\langle x^{(i)}\rangle, \langle y^{(i)}\rangle; \langle z^{(i)}\rangle\}_{i\in\mathbb{Z}_N})$ 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 $k = 1, \dots, R$, all parties perform:
    – $\{\{\langle x^{(i)}\rangle^{\ell[x]}, \langle y^{(i)}\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_{N/2^k}}; \langle z\rangle^{\ell[x]}\} \leftarrow$
      $\Pi_{\mathsf{Reduce}}(\{\langle x^{(i)}\rangle^{\ell[x]}, \langle y^{(i)}\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_{N/2^{k-1}}}; \langle z\rangle^{\ell[x]})$;
- All parties invoke
  $b = \Pi_{\mathsf{InnerVerify}}(\{\langle x^{(i)}\rangle^{\ell[x]}, \langle y^{(i)}\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_{N/2^R}}; \langle z\rangle^{\ell[x]})$;
- All parties output $b$.
```

Figure 7: The Batch Multiplication Verification Protocol

Compression of multiplication triples. We first design a subprotocol Π_{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 $\{\langle x^{(i)}\rangle^{\ell[x]}, \langle y^{(i)}\rangle^{\ell[x]}, \langle z^{(i)}\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_N}$ locally, which is over the polynomial ring. 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 zero shares. 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 Π_{Rec}. All parties locally calculate $\langle z\rangle^{\ell[x]} = \sum_{i=0}^{N-1} r^i \cdot \langle z^{(i)}\rangle^{\ell[x]}$, and $\langle x'^{(i)}\rangle^{\ell[x]} = r^i \cdot \langle x^{(i)}\rangle^{\ell[x]}$ for all $i \in \mathbb{Z}_N$ 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* Π_{Trans} *take* $\{\langle x^{(i)}\rangle, \langle y^{(i)}\rangle, \langle z^{(i)}\rangle\}_{i\in\mathbb{Z}_N}$ *as input, 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.3. □

Dimension reduction. We extend the dimension reduction technique of Goyal *et al.* [22] to our 3PC over ring setting. As shown in Fig. 5, protocol Π_{Reduce} takes a shared triple $(\{\langle x^{(i)}\rangle^{\ell[x]}, \langle y^{(i)}\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_N}, \langle z\rangle^{\ell[x]})$ as input and outputs $(\{\langle x'^{(i)}\rangle^{\ell[x]}, \langle y'^{(i)}\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_{N/2}}, \langle z'\rangle^{\ell[x]})$. Π_{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 $h(1) = z - h(0)$. 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 $5\ell \cdot d$ 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* Π_{Reduce} *take* $(\{\langle x^{(i)}\rangle^{\ell[x]}, \langle y^{(i)}\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_N}, \langle z\rangle^{\ell[x]})$ *as input, and it outputs the new list* $(\{\langle x'^{(i)}\rangle^{\ell[x]}, \langle y'^{(i)}\rangle^{\ell[x]}\}_{i\in\mathbb{Z}_{N/2}}, \langle z'\rangle^{\ell[x]})$. *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.4. □

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 $\sum_{i=0}^{N/2^R} \langle x^{(i)}\rangle^{\ell[x]} \cdot \langle y^{(i)}\rangle^{\ell[x]} = \langle z\rangle^{\ell[x]}$, $\Pi_{\mathsf{InnerVerify}}$ 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, which requires $(3N/2^R+1)\ell \cdot d$ bit ($3N/2^R+1$ cross-items) communication in the offline phase and $5\ell d$ bits ($2\ell d$ for revealing m_z, $3\ell d$ for zero check). 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* $z \neq \sum_{i=0}^{N-1} x^{(i)} \cdot y^{(i)}$ *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.5. □

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

P_j and P_k hold the common seed $\eta_{j,k} \in \{0,1\}^{\lambda}$.
Input : $\langle\cdot\rangle$-shared value of x.
Output : P_j for $j \in \mathbb{Z}_3$ output v_j. $v_0 = v_1 = v_2$, if the sign bit of x is 0.

Execution:

- Parse $\langle x \rangle := \{m_x, [r_x]_1, [r_x]_2\}$ as $\{x_0, -x_2, -x_1\}$ where P_j for $j \in \mathbb{Z}_3$ holds x_{j-1} and x_{j+1};
- The verifier P_k calculates $r = x_{k-1} + x_{k+1} - \mathsf{sign}(x_{k-1} + x_{k+1}) \cdot 2^{\ell-1}$. Then P_k chops $2^{\ell-1} - r$ as $\{r_0, \dots, r_{\ell-2}\}$;
- All parties performs $\|r_i\| \leftarrow \Pi^{p,k}_{\|\cdot\|}(r_i)$ for $i \in \mathbb{Z}_{\ell-1}$, taking the biggest prime of $p \in (\ell, 2^{\log \ell+1}]$;
- P_{k-1} and P_{k+1} do:
  1) set $\|r_{\ell-1}\| = \|0\|$;
  2) pick a random value $\Delta \in \{0,1\}$ with seed $\eta_{k-1,k+1}$.
  3) pick a random list $\{w_i, w'_i\}_{i \in \mathbb{Z}_\ell} \in (\mathbb{Z}_p^*)^{2\ell}$ with seed $\eta_{k-1,k+1}$;
  4) pick a random permutation π with seed $\eta_{k-1,k+1}$;
  5) set $s = x_k - \mathsf{sign}(x_k) \cdot 2^{\ell-1}$, bit-exact it as $\{s_i\}_{i \in \mathbb{Z}_{\ell-1}}$ and set $s_{\ell-1} = 0$;
  6) calculate $\|m_i\| = s_i + \|r_i\| - 2s_i \cdot \|r_i\|$ and $\|m'_i\| = \sum_{t=0}^{i} \|m_t\| - 2 \cdot \|m_i\| + 1$ for $i \in \mathbb{Z}_\ell$;
  7) calculate $\|u_i\| = \pi(w_i \cdot \|m'_i\| + m_{\sigma,i} \oplus \mathsf{sign}(x_i) \oplus \Delta)$ and $\|u'_i\| = \pi(w'_i(w_i \cdot \|m'_i\| - (p-1)))$ for $i \in \mathbb{Z}_\ell$;
  8) output $v_{k-1} = \Delta$ or $v_{k+1} = \Delta$.
- All parties invoke $u_i = \Pi^{p,k}_{\mathsf{Rec}}(\|u_i\|)$ and $u'_i = \Pi^{p,k}_{\mathsf{Rec}}(\|u'_i\|, P_i)$ for $i \in \mathbb{Z}_\ell$. Consequently, P_k holds $\{u_i, u'_i\}_{i \in \mathbb{Z}_\ell}$.
- P_k output $v_k = \mathsf{sign}(x_{k-1} + x_{k+1})$ if $\exists u_i = 0 \wedge u'_i \neq 0$ for $i \in \mathbb{Z}_{\ell+1}$, otherwise P_k output $v_k = 1 \oplus \mathsf{sign}(x_{k-1} + x_{k+1})$.

Figure 8: Positive Verification Protocol Verified by P_k.

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 $(5R+5)\ell \cdot d$ 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. 3.

Theorem 3. *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.6. □

Protocol $\Pi_{\mathsf{Rec}}(\{\|x^{(i)}\|\}_{i \in \mathbb{Z}_N})$

Input : N numbers of $\|\cdot\|$-shared value.
Output : P_0 receive $\{x^{(i)}\}_{i \in \mathbb{Z}_N}$.

Execution:

- P_1 and P_2 reveal $\{x^{(i)}\}_{i \in \mathbb{Z}_N}$ to P_0;
- P_0 picks λ random value $\{w_k \in \mathbb{Z}_p^*\}_{k \in \mathbb{Z}_\lambda}$ and send them to P_1 and P_2.
- P_1 and P_2 do
  - calculate $[\![t_{k,k'}]\!] = \sum_{i=0}^{N-1} w_k^i \cdot [\![\gamma(x^{(i)})_{k'}]\!]$ for $k \in \mathbb{Z}_\lambda, k' \in \mathbb{Z}_\lambda$.
  - reveal $\{t_{k,k'}\}_{j \in \mathbb{Z}_\lambda, k \in \mathbb{Z}_\lambda}$ to P_0.
- P_0 caclualte $\hat{t}_{k,k'} = \alpha_{k'} \cdot \sum_{i=0}^{N-1} w_k^i \cdot x^{(i)}$ and abort if exist $\hat{t}_{k,k'} \neq t_{k,k'}$ for any $k' \in \mathbb{Z}_\lambda, k \in \mathbb{Z}_\lambda$.

Figure 9: The Batch Verifiable Reconstruction Protocol

4.2. Sign Bit Extraction Batch Verification

In this section, we upgrade the sign bit extraction Π_{SignBit} to the malicious setting throughout the verification protocol. For a sign bit extraction pair $\{\langle x \rangle, \langle z \rangle\}$ s.t. $z = \mathsf{sign}(x)$, the malicious adversary can introduce arbitrary errors to make $\mathsf{sign}(x) \neq z$. As shown in Fig. 10, we design the verification protocol Π_{VSignBit} to verify the correctness of the sign bit extraction pair.

One-bit leakage model. Notice that a malicious adversary can introduce a probabilistic error based on the input. For instance, the corrupted P_0 can introduce an error to the input x when P_0 generates $[\![r_{x,i}]\!]$ in step (3) of the preprocessing phase of Fig. 2. Therefore, during the verification procedure, P_0 is able to launch a selective failure attack. That is, P_0 introduces an error e to the input x. Depending on the verification result, the adversary can judge whether $x + e$ changes the sign bit. Similarly, the corrupted P_1 or P_2 can also introduce errors on m_x while calculating list u_i and u'_i. To mitigate such a leakage, we design a batch verification protocol that combines all verification into a single check, which reduces the overall leakage of large batch size N to one bit. At the end of this section, we formalize this leakage in functionality $\mathcal{F}_{\mathsf{VSignBit}}$ and prove the security of Π_{VSignBit}.

Specifically, our sign bit extraction verification consists of two steps: (i) z is validated to be either 0 or 1, (ii) $x - 2^{\ell-1} \cdot z$ is positive. The former check can be realized by employing a maliciously secure multiplication protocol to confirm that its square matches itself, i.e., $z \cdot z = z$ on the ring $\mathbb{Z}_{2^\ell}$, 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_k 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 [23], [26] 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_k (any $i \in$

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:

All parties do:

- Calculate $\langle x'^{(i)} \rangle = \langle x^{(i)} \rangle - 2^{\ell} \langle z^{(i)} \rangle$
- Call $\{v_0^{(i)}, v_1^{(i)}, v_2^{(i)}\}_{i \in \mathbb{Z}_N} = \Pi_{\mathsf{Pos}}(\{\langle x'^{(i)} \rangle\}_{i \in \mathbb{Z}_N}, P_1)$ and $\{v_0^{(i+N)}, v_1^{(i+N)}, v_2^{(i+N)}\}_{i \in \mathbb{Z}_N} = \Pi_{\mathsf{Pos}}(\{\langle x'^{(i)} \rangle\}_{i \in \mathbb{Z}_N}, P_2)$;
- P_j for $j \in \mathbb{Z}_3$ generate $t_j = H(v_j^{(1)} || v_j^{(2)} || \ldots || v_j^{(2N)})$ and share it as $\langle t_j \rangle$; where $H : \{0,1\}^* \mapsto \{0,1\}^{\ell}$ is a collision resistant hash function.
- All parties invoke $\langle \alpha_j \rangle \leftarrow \Pi^{\ell}_{\langle \cdot \rangle}$ for $j \in \mathbb{Z}_3$.
- All parties call $\Pi^{R}_{\mathsf{MultVerify}}$ with $\{\langle z^{(i)} \rangle, \langle z^{(i)} \rangle; \langle z^{(i)} \rangle\}_{i \in \mathbb{Z}_N}$ and $\{\langle \alpha_j \rangle, \langle t_{j-1} - t_{j+1} \rangle; \langle 0 \rangle\}_{j \in \mathbb{Z}_3}$;
- If no party aborts, all parties output 1.

Figure 10: The Sign Bit Extraction Verification Protocol.

$\{0,1,2\}$) take input as shared value $\langle x \rangle$, and the verifier outputs a bit indicating whether $2^{\ell-1} \overset{?}{\geq} x$. Specifically, we introduce the IT-secure MAC to detect malicious behavior of P_{k-1} and P_{k+1}. We observe that the chopped shared bit $[\![r_{x,i}]\!]$ in Π_{SignBit} can be replaced by $\|r_{x,i}\|$. We let the presumably honest verifier P_k locally calculate the λ MACs of $r_{x,i}$ and secret share it to the other two parties P_{k-1} and P_{k+1}. Later, when P_{k-1} and P_{k+1} send back the opened vector $\{\|u_i\|^{p,\lambda,k}\}_{i \in \mathbb{Z}_\ell}$ and $\{\|u_i'\|^{p,\lambda,k}\}_{i \in \mathbb{Z}_\ell}$, P_k can check the correctness of them by the corresponding MAC. For the batch verification, we let P_k not reshare the aforementioned $z \oplus \Delta$. Considering the positive sign bit ($z = 0$), we have $z \oplus \Delta = \Delta$, where P_k holds $v_k = z \oplus \Delta$, P_{k+1} holds $v_{k+1} = \Delta$ and P_{k-1} holds $v_{k-1} = \Delta$. The positive assertion protocol is converted to verify $v_0 = v_1 = v_2$. We introduce the batch equality test to address this problem. The soundness error of the verifier P_0 in $\Pi^{p,\lambda}_{\mathsf{Pos}}$ is $\frac{1}{2^{\lambda \log \ell + \lambda + \log \ell}}$.

Theorem 4. *Let $\langle x \rangle^{\ell}$ be the input of the protocol $\Pi^{p,\lambda}_{\mathsf{Pos}}$ depicted Fig. 8. The probability that output of $\Pi^{p,\lambda}_{\mathsf{Pos}}$ for each party is not equal and $\mathsf{sign}(x) = 1$ is at most $\frac{1}{2^{\lambda \log \ell + \lambda + \log \ell}}$.*

Proof. See Appendix A.7. ☐

Dual execution. To support the 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 $\hat{r}_x = -r_x - \mathsf{sign}(-r_x) \cdot 2^{\ell-1}$ with $r = x_{k-1} + x_{k+1} - \mathsf{sign}(x_{k-1} + x_{k+1})$ to generate the the vector $\{\|u_i\|\}_{i \in \mathbb{Z}_\ell}$ and $\{\|u_i'\|\}_{i \in \mathbb{Z}_\ell}$. With dual execution Π_{Pos} with N pairs sign bit, each party P_j hold $v_j^{(i)}$ for $i \in \mathbb{Z}_{2N}$.

Batch equality test. To mitigate one-bit leakage, we need to combine all verification output in a single check. The verification protocol requires to satisfy two properties: (1) Overall revealed information is only $v_0^{(i)} = v_1^{(i)} = v_2^{(i)}$ without any intermediate information such as $v_0^{(i)} = v_1^{(i)}$. Any intermediate information will allow the adversary to get another 1-bit information leakage; (2) The corrupted party P_j can not affect the verification of $v_{j-1}^{(i)} = v_{j+1}^{(i)}$. This property is designed for soundness, which ensures that the other two parties detect the correctness of the result through their shares. Fig. 10 depicts the procedure of the batch equality test. $H : \{0,1\}^* \mapsto \{0,1\}^{\ell}$ is a collision resistant hash function. For $2N$ numbers of output $\{v_0^{(i)}, v_1^{(i)}, v_2^{(i)}\}$ for $i \in \mathbb{Z}_{2N}$, if the output is correct, let $t_j = H(v_j^{(1)} || v_j^{(2)} || \ldots || v_j^{(2N)})$ for $j \in \mathbb{Z}_3$, we have $t_0 = t_1 = t_2$. To verify $t_0 = t_1 = t_2$, we let all parties generate $\langle \alpha_j \rangle$ and invoke $\Pi^{R}_{\mathsf{MultVerify}}(\langle \alpha_j \rangle, \langle t_{j-1} - t_{j+1} \rangle; \langle 0 \rangle)$. We analyze the security as follows: (1) If P_1 (or P_2) is corrupted, the error introduced by P_1 when performing Π_{Pos} with verifier P_2 will be captured by MAC check. Therefore, the output of Π_{Pos} with verifier P_2 held by P_0 and P_2 (corresponding to $v_0^{(i)}$ and $v_2^{(i)}$) is correctly calculated. For this part, $\Pi^{R}_{\mathsf{MultVerify}}$ can verify the parts of $v_0^{(i)} = v_2^{(i)}$ which corresponding to Π_{Pos} with verifier P_2. (2) If P_0 is corrupted, Π_{Pos} with verifier P_1 or P_2 can both calculate the positive of $x - 2^{\ell-1} \cdot z$ correctly, cause the error introduced by P_0 in Π_{Pos} will be captured by MAC check and the result of Π_{Pos} can be calculated using $\Pi^{R}_{\mathsf{MultVerify}}$ with verifying $v_1^{(i)} = v_2^{(i)}$. We observe that the $\Pi_{\mathsf{MultVerify}}(\langle z \rangle, \langle z \rangle, \langle z \rangle)$ will leak another 1-bit leakage, causing the adversary can introduce error -1 or 1 on the output z and infer the value of z depending on the verification result. We combine this verification with $\Pi^{R}_{\mathsf{MultVerify}}(\langle \alpha_j \rangle, \langle t_{j-1} - t_{j+1} \rangle; \langle 0 \rangle)$, which pack the overall leak information as $\bigwedge_i^{2N} (v_0^{(i)} = v_1^{(i)} = v_2^{(i)}) \wedge \bigwedge_i^{N} z_i \in \{0,1\}$.

Batch MAC Verification. We observe that the batch MAC verification can be used to reduce the reconstruction communication further. For N pairs of $\| \cdot \|$-shared value $\|x^{(0)}\|, \ldots, \|x^{(N-1)}\|$, P_1 and P_2 partially open secret value $x^{(i)}$ (without the MACs) to P_0. We let P_0 generate a public λ-dimension random list $\{w_k \in \mathbb{Z}_p\}_{k \in \mathbb{Z}_\lambda}$ and send the list to P_1 and P_2. With the random list, the N pairs of MACs can be combined to λ pairs, that is, $\|t_k\| = \sum_{i=0}^{N-1} w_k^i \cdot \|x^{(i)}\|$ for $k \in \mathbb{Z}_\lambda$. Instead of verifying n pairs of share, P_0 only needs to verify $\alpha \cdot t_k = \gamma(t_k)$ for $k \in \mathbb{Z}_\lambda$, where $n \gg \lambda$. Note that, the batch MAC verification requires an additional round for the MAC opening. Combining with batch MAC verification, our positive assertion protocol Π_{Pos} requires 2-round communication of $(\lambda + 1)(\ell - 1) \log \ell + 4\ell \log \ell$ bits, where λ is MAC key number of $\| \cdot \|$.

Security. We define the functionality $\mathcal{F}_{\mathsf{VSignBit}}$ for sign bit extraction in the 1-bit leakage model in Fig. 11. Before outputting the result, it receives a boolean function f from the adversary and reveals the function evaluation result to the adversary, which leaks 1-bit information.

Theorem 5. *Let $\mathsf{PRF}^{\mathbb{Z}_p}$ and $\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}$ be the secure pseudo-random functions. The protocol Π_{VSignBit} as depicted in*

Functionality $\mathcal{F}^{N}_{\mathsf{VSignBit}}[\mathbb{Z}_{2^\ell}]$

$\mathcal{F}_{\mathsf{SignBit}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. Let $\mathcal{C}_{\mathsf{cor}}$ denote the set of corrupted parties.

Input:

- Upon receiving $(\mathsf{Input}, \mathsf{sid}, \left\{(r_1^{(i)}, r_2^{(i)})\right\}_{i\in\mathbb{Z}_N})$ from P_0, send $(\mathsf{Input}, \mathsf{sid}, P_0)$ to $\mathcal{S}$ and record $\left\{(r_1^{(i)}, r_2^{(i)})\right\}_{i\in\mathbb{Z}_N}$;
- Upon receiving $(\mathsf{Input}, \mathsf{sid}, \left\{(m_j^{(i)}, r_j'^{(i)})\right\}_{i\in\mathbb{Z}_N})$ from P_j, $j \in \mathbb{Z}_2$, send $(\mathsf{Input}, \mathsf{sid}, P_j)$ to $\mathcal{S}$ and record $\left\{(m_j^{(i)}, r_j'^{(i)})\right\}_{i\in\mathbb{Z}_N}$;

Execution:

- If $\exists i \in \mathbb{Z}_N, \exists j \in \{1,2\} : r_j^{(i)} \neq r_j'^{(i)}$, send $(\mathsf{Output}, \mathsf{sid}, \perp)$ to P_0, P_1 and P_2; halt;
- Let $j^* \in \{1,2\}$ be an index s.t. $P_{j^*} \in \mathcal{P} \setminus \mathcal{C}_{\mathsf{cor}}$. It computes $z^{(i)} := \mathsf{sign}(m_{j^*}^{(i)} - r_1^{(i)} - r_2^{(i)})$, for $i \in \mathbb{Z}_N$;
- Upon receiving $(\mathsf{Compute}, \mathsf{sid}, \mathsf{Alg}, f)$ from $\mathcal{S}$, compute $(z'^{(0)}, \dots z'^{(N-1)}) \leftarrow \mathsf{Alg}(\left\{(r_1^{(i)}, r_2^{(i)}, m_1^{(i)}, m_2^{(i)})\right\}_{i\in\mathbb{Z}_N})$ and $b \leftarrow f(\left\{(r_1^{(i)}, r_2^{(i)}, m_1^{(i)}, m_2^{(i)})\right\}_{i\in\mathbb{Z}_N})$;
- If $b \in \{0,1\}$, send $(\mathsf{Leak}, \mathsf{sid}, b)$ to $\mathcal{S}$;
- If $\exists i \in \mathbb{Z}_N : z^{(i)} \neq z'^{(i)}$, send $(\mathsf{Output}, \mathsf{sid}, \perp)$ to P_0, P_1 and P_2; halt;
- For $i \in \mathbb{Z}_N$, pick random $u_1^{(i)}, u_2^{(i)} \leftarrow \mathbb{Z}_{2^\ell}$, set $u^{(i)} := u_1^{(i)} + u_2^{(i)}$ and $w^{(i)} := z'^{(i)} + u^{(i)}$;
- Send $(\mathsf{Output}, \mathsf{sid}, \{(u_1^{(i)}, u_2^{(i)})\}_{i\in\mathbb{Z}_N})$ to P_0, $(\mathsf{Output}, \mathsf{sid}, \{(w^{(i)}, u_1^{(i)})\}_{i\in\mathbb{Z}_N})$ to P_1, $(\mathsf{Output}, \mathsf{sid}, \{(w^{(i)}, u_2^{(i)})\}_{i\in\mathbb{Z}_N})$ to P_2.

Figure 11: The ideal functionality $\mathcal{F}^{N}_{\mathsf{VSignBit}}[\mathbb{Z}_{2^\ell}]$.

Fig. 10 UC realizes $\mathcal{F}_{\mathsf{VSignBit}}$ against malicious PPT adversaries who can statically corrupt up to one party.

Proof. See Appendix A.8. □

Our Sign bit extraction protocol Π_{VSignBit} requires amortized 2-round communication of $2((\lambda+1)(\ell-1)\log\ell + 6\ell\log\ell + \ell)$ bits, where λ is MAC key number of $\|\cdot\|$.

5. The Aegis PPML Platform

Through the above arithmetic/non-arithmetic components, we can construct our privacy-preserving machine learning platform Aegis. We give a brief introduction to facilitate the reader's understanding of the working mechanism of our PPML components. For the protocol details, we refer the reader to further read Appendix. B.

Arithmatic protocol. Our maliciously secure multiplication protocol is shown in Fig. 12. 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. The multiplication protocol can be expanded to the inner product protocol (Cf. Appendix. B.1). At the high level, all parties combine multiple inner product triples to single inner product triples and perform similar dimension reduction, which also reduces to the sublinear communication cost. For the matrix multiplication and convolution, we view them as multiple separate inner products.

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 $\Pi^{R}_{\mathsf{MultVerify}}(\{\langle x^{(i)}\rangle, \langle y^{(i)}\rangle; \langle z^{(i)}\rangle\}_{i\in\mathbb{Z}_N})$ to verify correctness.

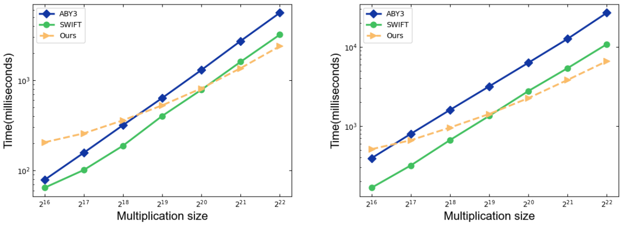
Figure 12: The Multiplication Protocol

Non-Arithmatic protocol. We propose a maliciously secure probabilistic trucation protocol (Cf. Appendix. B.2), which is used to reduce the 2^k scaler caused by fixed-point multiplication. Our idea is derived from SWIFT [27] which generates correct truncation pair via maliciously secure inner product protocol.

Secure ReLU Protocol. 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} (Cf. Appendix. B.3, Fig. 22) 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:

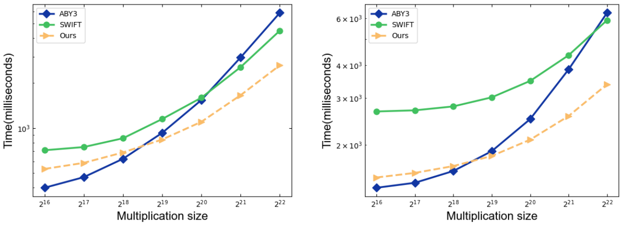
$$\begin{aligned} m_w &= m_x m_z + m_x r_z + m_z r_x + r_x r_z - r_w \\ &= m_x m_z + m_x r_z + (m' - 2\Delta m' + \Gamma) r_x + r_x r_z - r_w \\ &= m_x m_z + m_x r_z + (1 - 2\Delta)(m' r_x + r'') + \Gamma' \end{aligned}$$

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



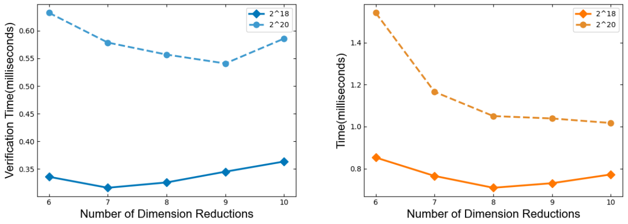
(a) MAN (b) WAN

Figure 13: Overall running time of multiplication (over the GPU setting). Compared with ABY3 [30], SWIFT [27] of Π_{Mult} over MAN and WAN setting.



(a) 32 Depth (b) 128 Depth

Figure 14: Evaluate the multiplication (over the GPU setting) with circuit depth 32 and 128 under the MAN setting.



(a) MAN (b) WAN

Figure 15: The running time of verification phase (over the GPU setting), with the different dimension reduction number R, multiplication triple size 2^{18} and 2^{20}, over MAN and WAN setting.

of m_w, with the fresh random value r''. Our ReLU protocol requires 1 rounds and communication of $(\ell-1)\log\ell+2\ell$ bits in the preprocessing phase and requires 2 rounds and communication of $4\ell\log\ell+4\ell$ bits in the online phase. 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.

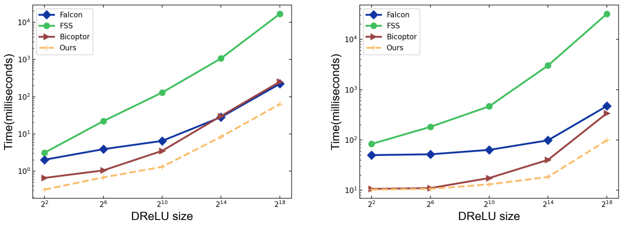
Secure Maxpool protocol. Our Maxpool scheme is constructed by comparison $\mathsf{great}(x,y) = x \overset{?}{\geq} y$ and maximum $\mathsf{max}(x_1,\ldots,x_n)$. In the case of signed numbers x and y, $\mathsf{great}(x,y)$ can be implemented by invoking the Π_{VSignBit} three times. That is, $\mathsf{great}(x,y) = (\mathsf{sign}(x)\oplus\mathsf{sign}(y))\cdot\mathsf{sign}(y-x)+(1\oplus\mathsf{sign}(x)\oplus\mathsf{sign}(y))\cdot\mathsf{sign}(y)$. 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 $\mathsf{max}(x_1,\ldots,x_n)$. One is to evaluate $\mathsf{max}(x_1,\ldots,x_n)$ by $\mathsf{max}(x_1,\ldots,x_n) = \sum_{i=1}^{n}(\Pi_{j=1,j\neq i}^{n}\mathsf{great}(x_i,x_j)\cdot x_i)$, 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 $\mathsf{max}(x_1,\ldots,x_n) = \mathsf{max}(\mathsf{max}(x_1,x_2),\ldots,v(x_{n-1},x_n))$. 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 $z_{i,j} = \mathsf{max}(x_{i,j},x_{i,j+1},x_{i+1,j},x_{i+1,j+1})$ and $z_{i,j+1} = \mathsf{max}(x_{i,j+1},x_{i,j+2},x_{i+1,j+1},x_{i+1,j+2})$. 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 $(n-1)((\ell-1)\log\ell+2\ell)$ bits of communication cost in the setup phase and $(n-1)(4\ell\log\ell+4\ell)$ bits in the online phase. Analogously, the malicious version of Maxpool can be achieved through verifying sign bit-exact and multiplication respectively.

Security. Assume our Aegis platform accepts inputs from P_i for $i\in\mathbb{Z}_3$ and invokes multiple times of semi-honest secure protocols (which can also ensure the privacy against malicious adversaries) and perform an overall maliciously secure verification of multiplication and sign bit extraction. We analyze the overall leakage of Aegis as follows. (i)For the execution phase, our protocol will leak no information, cause it can ensure privacy against malicious adversaries. (ii) For the verification phase, the potential leakage is caused by the times of verification. As mentioned before, our sign-bit verification protocol (Cf. Fig. 10) is reduced to $\Pi^R_{\mathsf{MultVerify}}$. Considering that multiplication verification protocol (Cf. Fig. 10) is also reduced to $\Pi^R_{\mathsf{MultVerify}}$, Aegis combine all the invoking of $\Pi^R_{\mathsf{MultVerify}}$ to single invoking, while the overall reveal message is one bit.

6. Implementation and Benchmarks

In this section, we evaluate our multiplication and non-arithmetic protocols in both the semi-honest and malicious settings. For the maliciously secure multiplication protocols, we compare the communication and runtime with SWIFT [27] and ABY [30]. For the non-arithmetic protocols, we compare the runtime performance with Bicoptor [43], BLAZE [33], SWIFT, FSS [5], Falcon [38] respectively.



(a) LAN

(b) MAN

Figure 16: The overall running time (over the CPU setting) of DReLU(Π_{SignBit}). Compared with Falcon [38], FSS [5], Bicoptor [43], over LAN and MAN settings.

Benchmark setting. We perform our arithmetic protocols on the GPU setting. To support GPU, our code is based on the Piranha [39] source code [40] which is a GPU platform for MPC protocols. For the non-arithmetic protocols, we implement both CPU and GPU versions to support benchmarking with FSS [5] and garble circuit-based protocol BLAZE [33] on CPU setting. In our benchmark setting, we take the size of the ring $\ell = 64$ and the polynomial ring degree $d = 64$. 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: 80ms, 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, 4× Nvidia 2080 Ti with 11 GB RAM and 1TB SSD.

Multiplication. We compare our maliciously secure multiplication protocol with SOTA. We benchmark the communication of Π_{Mult} and Π_{Inner} in the Appendix C.2 and the running time in Fig. 13. For the running time, we execute the protocol at multiple R values, choosing the best performance. Influenced by an additional verification round which is the dominant overhead in the case of a small volume of data, our protocol is worse than SWIFT and ABY. Considering saturated data, our protocol achieves 2× the performance improvement compared to SWIFT and ABY under both MAN and WAN settings. Considering the multiplication depth, Fig. 14 shows the performance changes under different multiplication depths. We benchmark protocols on multiplication circuits with depths of 32 and 128. Since our protocol and ABY can ensure round advantages based on batch verification, the performance is better than the SWIFT protocol when the multiplication depth is large.

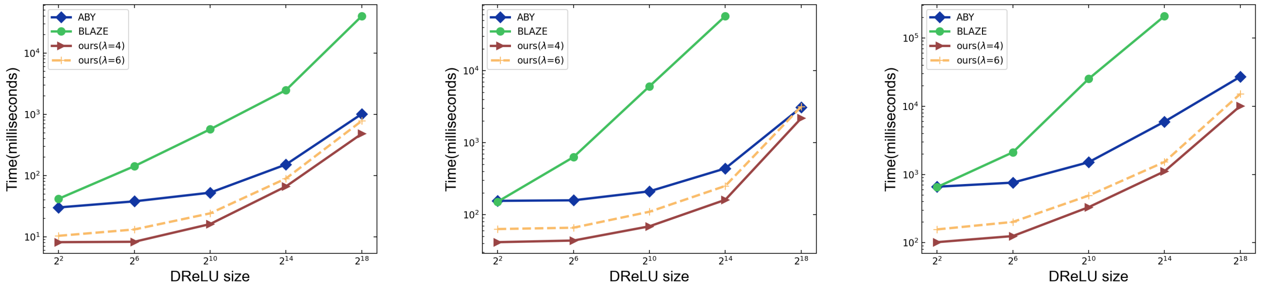
Trade-off of the repetition parameter R. While selecting a larger value for the repetition parameter R for dimension reduction can minimize the communication volume in batch verification, it is also essential to consider the impact of additional communication rounds in the postprocessing phase for overall performance. We conduct a practical experimental benchmark to determine the optimal value of R in different bandwidth and delay scenarios. Fig. 15 depicts the verification time with the different dimension reduction number R. It points out the optimal R value ($R = 7$ in MAN, with data size 2^{18}; $R = 9$ in MAN, with data size 2^{20}; $R = 8$ in WAN, with data size 2^{18};$R = 10$ in WAN, with data size 2^{20};). Our benchmark indicates that the larger R needs to be chosen for smaller bandwidths and larger data dimensions.

Non-arithmetic functions. The benchmark data in Fig.16 and Fig.17 demonstrates the high efficiency of our nonlinear protocol. Fig.16 depicts the overall running time comparison of the semi-honest secure ReLU protocol (over the CPU setting) with SOTA [5], [38], [43] in LAN, and MAN settings (For Bicoptor, we take the truncation error parameter $\ell^* = 32$). Fig.17 depicts the overall running time comparison of the maliciously secure ReLU protocol (over the CPU setting) with SOTA [27], [30], [33] in LAN, MAN and WAN settings. 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 2^2 to 2^{18}. 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 ($\lambda = 4$ for soundness error 2^{-34} and $\lambda = 6$ for soundness error 2^{-48}). Under the semi-honest threat model and WAN setting, as anticipated, our semi-honest protocol demonstrates a performance improvement of 4× compared to the constant round protocol Bicoptor (theoretically, communication volume has been reduced by 4× on a 64-bit ring). Under the malicious threat model, compared to the constant round protocol BLAZE, our maliciously secure version achieves over 100× performance improvement with a reasonable ReLU size. Since the delay dominates the execution overhead considering the small amount of data, our 2-round protocol is much lower than the logarithmic rounds protocol ABY in terms of time cost. In the above cases, the performance of our protocol is more than 4× that ABY, no matter in LAN, MAN, or WAN settings. For the WAN setting, the performance of our protocol is more than 6× that ABY considering the small batches of input. The performance of our protocols under a semi-honest setting is provided in Appendix C.3. We also compare our semi-honest protocol with Piranha [39] over the GPU setting((Cf. Appendix. C.4)), where our protocol achieves more than 3× performance improvement compared to Piranha.

The inference of neural network. We benchmark the inference of the neural network based on Piranha (Cf. Appendix. C.1). For more benchmark results, we refer the reader to Appendix. C.

7. 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



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

Figure 17: Overall run-time of ReLU in LAN/MAN/WAN setting. Here, "ours" refers to our maliciously secure protocols (Soundness error 2^{-48} for $\lambda = 6$ and 2^{-34} for $\lambda = 4$); BLAZE refers to [33]; ABY refers to [30].

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., [27], [30], [33], [43].

References

[1] Pierre Baldi, Roberta Baronio, Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Countering gattaca: efficient and secure testing of fully-sequenced human genomes. In *CCS*, 2011.

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

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

[4] 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.

[5] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. Function secret sharing for mixed-mode and fixed-point secure computation. In *EUROCRYPT*, 2021.

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

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

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

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

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

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

[12] 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.

[13] 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.

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

[15] Anders Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-Majority Four-Party secure computation with malicious security. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

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

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

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

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

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

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

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

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

[24] 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.

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

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

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

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

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

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

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

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

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

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

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

[36] 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.

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

[38] 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.

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

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

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

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

[43] 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* $\mathcal{L} := (L_0, \dots, L_{\ell-1}) \in \{0,1\}^\ell$ *be a binary vector. There exists a linear transformation* ϕ *such that* $\phi(\mathcal{L}) = (L'_0, \dots, L'_{\ell-1})$ *satisfies:*

- Let $i^* \in \mathbb{Z}_\ell$ be the index of the first non-zero bit in $\mathcal{L}$, that is, $L_{i^*} = 1 \ \wedge \ \forall i < i^* : L_i = 0$.
- $L'_{i^*} = 0$ and $L'_j \neq 0$ for all $i \neq i^*$.

Proof. Consider the transformation $\phi(\mathcal{L}) := (L'_0, \dots, L'_{\ell-1})$ such that $L'_i = \sum_{t=0}^{i} L_t - 2 \cdot L_i + 1$ for $i \in \mathbb{Z}_\ell$. Let $s_i := \sum_{t=0}^{i} L_t$ be the prefix-sum of $\mathcal{L}$ and $\mathcal{L}' = \phi(\mathcal{L}) = s_i - 2 \cdot L_i + 1$. We argue that $\mathcal{L}'$ will only contain one zero at the position i^*, where $L'_i \neq 0$ for all $i \neq i^*$. Indeed, it converts all the prefix zero bits of $\mathcal{L}$ to 1 (namely, if $s_i = 0 \wedge \mathcal{L}_i = 0$ then $\mathcal{L}'_i = 1$); it converts the first non-zero bit of $\mathcal{L}$ to 0 (namely, if $s_{i^*} = 1 \wedge \mathcal{L}_{i^*} = 1$ then $\mathcal{L}'_{i^*} = 0$); it converts the suffix bits to non-zero values (namely, in case $\mathcal{L}_i = 0$, $s_i \geq s_{i^*} + \mathcal{L}_i = 1$, we have $\mathcal{L}'_i = s_i - 2\mathcal{L}_i + 1 \geq 2$; in case $\mathcal{L}_i = 1$, $s_i \geq s_{i^*} + \mathcal{L}_i = 2$, we have $\mathcal{L}'_i = s_i - 2\mathcal{L}_i + 1 \geq 1$).

This concludes our proof. ☐

A.2. The proof of Theorem 2.

Theorem 2. *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. 2 UC realizes* $\mathcal{F}_{\mathsf{SignBit}}$ *against malicious PPT adversaries who can statically corrupt up to one party.*

Proof. To prove Thm. 5, 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 $\{[\![r_{x,i}]\!]_1^p\}_{i \in \mathbb{Z}_{\ell-1}}, [r']_1$ form corrupted P_0 to P_1, and $\{[\![r_{x,i}]\!]_2^p\}_{i \in \mathbb{Z}_{\ell-1}}, [r']_2$ form corrupted P_0 to P_2, $\mathcal{S}$ extracts $\hat{r}_x = 2^{\ell-1} - \sum_{i=0}^{\ell-2} 2^{\ell-2-i}([\![r_{x,i}]\!]_1^p + [\![r_{x,i}]\!]_2^p)$ and $r' = [r']_1 + [r']_2$;
- $\mathcal{S}$ picks random list $\{\hat{u}'_i\}_{i \in \mathbb{Z}_\ell}$ where $\hat{u}'_i \in \mathbb{Z}_p$ and sets another list $\{\hat{u}_i\}_{i \in \mathbb{Z}_\ell}$ as following steps:
  - For each i where $\hat{u}'_i = 0$, set $\hat{u}_i \leftarrow \{p-1, 0\}$.
  - For each i where $\hat{u}'_i \neq 0$, set $\hat{u}_i \leftarrow \mathbb{Z}_p^*$.
  - Let $\mathcal{I} := \{i \mid \hat{u}'_i \neq 0\}$. Pick random $\alpha \leftarrow \mathcal{I}$ to set $\hat{u}_\alpha \leftarrow \mathbb{Z}_2$.
  - Send $\{\hat{u}'_i\}_{i \in \mathbb{Z}_\ell}$ and $\{\hat{u}'_i\}_{i \in \mathbb{Z}_\ell}$ to P_0.
- Upon receiving m'_1 from corrupted P_0 to P_1 and m'_2 from corrupted P_0 to P_2, $\mathcal{S}$ does:
  - generate $[r_x]_1$ and $[r_x]_2$ with $\eta_{0,1}$ and $\eta_{0,2}$.
  - calculate $r_x = [r_x]_1 + [r_x]_2$.
  - For $j \in \{1, 2\}$, if $\exists \hat{u}_i = 0 \wedge \hat{u}'_i \neq 0$, set $\delta_j = (m'_j + r') - \mathsf{sign}(r_x)$, else set $\delta_j = (m'_j + r') - (1 \oplus \mathsf{sign}(r_x))$.
  - Calculate $r'_x = -\hat{r_x} - \mathsf{sign}(-r_x) \cdot 2^{\ell-1}$.
  - Send $(\mathsf{Input}, \mathsf{sid}, r'_x - [r_x]_2, [r_x]_2)$ to $\mathcal{F}_{\mathsf{SignBit}}$.
  - Send $(\mathsf{Modify}, \mathsf{sid}, 0, 0, \delta_1, 0, \delta_2, 0)$ 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 $\mathsf{Real}_{\Pi_{\mathsf{SignBit}}, \mathcal{A}, \mathcal{Z}}(1^\kappa)$.

Hybrid $\mathcal{H}_1$: It is the idea world execution $\mathsf{Ideal}_{\mathcal{F}, \mathcal{S}, \mathcal{Z}}(1^\kappa)$ which is same as $\mathcal{H}_0$ except that in $\mathcal{H}_1$, list $\hat{u}_i$ and $\hat{u}'_i$ reveal to P_0 are picked uniformly random instead of calculating from $w_i \cdot m'_i + (\mathsf{sign}(m_x) \oplus \hat{m}_{x,i} \oplus \Delta)$ and $w'_i(w_i \cdot m'_i + 1)$.

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

Proof. Considering the list $\hat{u}'_i$, we change it derived from PRF to uniform random witch takes the advantage $\ell \cdot \mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_p}}(1^\kappa, \mathcal{A})$. Considering the list $\hat{u}_i$, if $\hat{u}'_i = 0$, the real execution $u_i = \mathsf{sign}(m_x) \oplus \hat{m}_{x,i} \oplus \Delta - 1$ is random (with $\mathsf{PRF}^{\mathbb{Z}_2}$) from $\{p-1, 0\}$ due to the random value $\hat{m}_{x,i} \in \mathbb{Z}_2$; if $\hat{u}'_i \neq 0$, the distribution of real execution $u_i = w_i \cdot m'_i + \mathsf{sign}(m_x) \oplus \hat{m}_{x,i} \oplus \Delta$ is that "one element random (with $\mathsf{PRF}^{\mathbb{Z}_2}$) from $\{0, 1\}$, and others random (with $\mathsf{PRF}^{\mathbb{Z}_p^*}$) from $\mathbb{Z}_p^*$". Because there only exists one position i that $m'_i = 0$. Considering the permutation π derived from $\mathsf{PRF}^{\mathbb{Z}_p^p}$ in real execution witch substituted by uniform random, the overall advantage is $\epsilon < 2 \cdot \ell \cdot \mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_p}}(1^\kappa, \mathcal{A}) + \mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_p^p}}(1^\kappa, \mathcal{A})$. ☐

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 $[r']_1$ using PRF with seed $\eta_{0,1}$.
- $\mathcal{S}$ 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, $\mathcal{S}$ does
  - Generate $[r_z]_1$ using PRF with the seed $\eta_{0,1}$.
  - Calculate $\delta = [\Gamma]_1 - [r']_1 + 2\Delta \cdot [r']_1 - [r_z]_1$,
  - Calculate $\Gamma = [\Gamma]_1 + [\Gamma]_2$.
- $\mathcal{S}$ picks $\llbracket r_{x,i} \rrbracket_1 \leftarrow \mathbb{Z}_p$ for $i \in \mathbb{Z}_\ell$ and acts as P_0 to send it to P_1.
- Upon receiving $\{\llbracket \hat{u}_j \rrbracket_1\}_{j \in \mathbb{Z}_{\ell+1}^*}$ and $\{\llbracket \hat{u}'_j \rrbracket_1\}_{j \in \mathbb{Z}_{\ell+1}^*}$ from corrupted P_1 to P_0, $\mathcal{S}$ does.
  - Invoke PRF with $\eta_{1,2}$ to generate permutation π, $\{w_i, w'_i\}_{i \in \mathbb{Z}_\ell} \in (\mathbb{Z}_p^*)^{2\ell}$, $\Delta \in \mathbb{Z}_2$.
  - Calculate $\{\llbracket u_i \rrbracket_1\}_{i \in \mathbb{Z}_\ell} = \pi^-(\{\llbracket \hat{u}_i \rrbracket_1\}_{i \in \mathbb{Z}_\ell})$.
  - Calculate $\{\llbracket u'_i \rrbracket_1\}_{i \in \mathbb{Z}_\ell} = \pi^-(\{\llbracket \hat{u}'_i \rrbracket_1\}_{i \in \mathbb{Z}_\ell})$.
  - Calculate $\hat{m}'_{x,i}$ via $\{\llbracket u'_i \rrbracket_1\}_{i \in \mathbb{Z}_{\ell+1}}$, w_i, w'_i and $\llbracket r_{x,i} \rrbracket_1$
  - Calculate $s_i \oplus \hat{m}'_{x,i}$ via Δ and$\{\llbracket u_i \rrbracket_1\}_{i \in \mathbb{Z}_\ell}$.
  - Set $m_x = s_1 || \hat{m}'_{x,1} || \ldots || \hat{m}'_{x,\ell-1}$.
  - Act as the corrupted P_1 (P_2) to send $(\mathsf{Input}, \mathsf{sid}, m_x)$ to the external $\mathcal{F}_{\mathsf{SignBit}}$.
- $\mathcal{S}$ sets $\mathsf{Alg}(r_1, r_2, m_1, m_2)$ as
  - calculate $r = r_1 + r_2$;
  - calculate and bit-extract $-r - \mathsf{sign}(-r) \cdot 2^{\ell-1}$ as $\{r_0, \ldots, r_{\ell-1}\}$;
  - perform $\llbracket r_i \rrbracket^p \leftarrow \Pi^p_{\llbracket \cdot \rrbracket}(r_i)$;
  - follow Π_{SignBit} steps (1)-(5) to calculate $\llbracket u_i \rrbracket_2$ and $\llbracket u'_i \rrbracket_2$ with m_2.
  - follow Π_{SignBit} steps (1),(3),(4) with m_1 and (i) for Π_{SignBit} step (2), set $\hat{m}_{x,\ell} = \hat{m}'_{x,\ell}$ with extracted value $\hat{m}'_{x,\ell}$; (ii) for Π_{SignBit} step (5), calculate $\llbracket u_i \rrbracket_1^p = w_i \cdot \llbracket m'_i \rrbracket^p + (s_i \oplus \hat{m}_{x,i} \oplus \Delta)$ with extracted value s_i.
  - output $z = \Delta \oplus \mathsf{sign}(-r)$ if $\exists(\llbracket \hat{u}_i \rrbracket_1^p + \llbracket \hat{u}_i \rrbracket_2^p = 0) \wedge (\llbracket \hat{u}'_i \rrbracket_1^p + \llbracket \hat{u}'_i \rrbracket_2^p \neq 0)$, else $z = \Delta \oplus \mathsf{sign}(-r) \oplus 1$.
- $\mathcal{S}$ sends $(\mathsf{Compute}, \mathsf{sid}, \mathsf{Alg})$ to $\mathcal{F}_{\mathsf{SignBit}}$.
- $\mathcal{S}$ sends $(\mathsf{Modify}, \mathsf{sid}, \{0, 0, \delta, 0, \delta, 0\})$ to $\mathcal{F}_{\mathsf{SignBit}}$.
- Upon receiving (Output, m_z, $[r_z]_1$) from $\mathcal{F}_{\mathsf{SignBit}}$, $\mathcal{S}$ 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 $\mathsf{Real}_{\Pi_{\mathsf{SignBit}}, \mathcal{A}, \mathcal{Z}}(1^\kappa)$.

Hybrid $\mathcal{H}_1$: It is same as $\mathcal{H}_0$ except that in $\mathcal{H}_1$, $\llbracket r_{x,i} \rrbracket_1$ and $[\Gamma]_1$ are picked uniformly random instead of calculating from $r_{x,j}$, $\Delta + [r']_2 - 2\Delta \cdot [r']_2 + [r_z]_2$.

Hybrid $\mathcal{H}_2$: It is the idea world execution $\mathsf{Ideal}_{\mathcal{F}, \mathcal{S}, \mathcal{Z}}(1^\kappa)$ which is same as $\mathcal{H}_1$ except that in $\mathcal{H}_1$, m' send to P_1 is calculated by $(m_z - \Gamma)/(1 - 2\Delta)$ instead of $\mathsf{sign}(-r_x) - r'$.

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

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 $\epsilon = \ell \cdot \mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_p}}(1^\kappa, \mathcal{A}) + 2\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}}(1^\kappa, \mathcal{A})$ by hybrid argument via reduction. ☐

This concludes the proof. ☐

A.3. The proof of Lemma 1.

Lemma 1. *Suppose protocol* Π_{Trans} *take* $\{\langle x^{(i)} \rangle, \langle y^{(i)} \rangle, \langle z^{(i)} \rangle\}_{i \in \mathbb{Z}_N}$ *as input, 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 $\frac{2^{(\ell-1)d}N+1}{2^{\ell d}} \approx \frac{N}{2^d}$. ☐

A.4. The proof of Lemma 2.

Lemma 2. *Suppose* Π_{Reduce} *take* $(\{\langle x^{(i)} \rangle^{\ell[x]}, \langle y^{(i)} \rangle^{\ell[x]}\}_{i \in \mathbb{Z}_N}, \langle z \rangle^{\ell[x]})$ *as input, and it outputs the new list* $(\{\langle x'^{(i)} \rangle^{\ell[x]}, \langle y'^{(i)} \rangle^{\ell[x]}\}_{i \in \mathbb{Z}_{N/2}}, \langle z' \rangle^{\ell[x]})$*. 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.5. 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.6. The proof of Theorem 3.

Theorem 3. *Let* $\{\langle x^{(i)}\rangle, \langle y^{(i)}\rangle, \langle z^{(i)}\rangle\}_{i\in\mathbb{Z}_N}$ *be the input of protocol* $\Pi_{\mathsf{MultVerify}}^{R}$ *depicted in Fig. 7. The probability* $\Pi_{\mathsf{MultVerify}}^{R}$ *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.7. The proof of Theorem 4.

Theorem 4. *Let* $\langle x\rangle^\ell$ *be the input of the protocol* $\Pi_{\mathsf{Pos}}^{\lambda}$ *depicted Fig. 8. The probability that* $\Pi_{\mathsf{Pos}}^{\lambda}$ *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 $\Pi_{\mathsf{Pos}}^{\lambda}$, 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.8. The proof of Theorem 5.

Theorem 4. *Let* $\mathsf{PRF}^{\mathbb{Z}_p}$ *and* $\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}$ *be the secure pseudorandom functions. The protocol* Π_{VSignBit} *as depicted in Fig. 10 UC realizes* $\mathcal{F}_{\mathsf{VSignBit}}$ *against malicious PPT adversaries who can statically corrupt up to one party.*

Proof. To prove Thm. 5, 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 $\mathcal{A}$.

- $\mathcal{S}$ extracts $r_x'^{(i)}$,$\delta_1^{(i)}$,$\delta_2^{(i)}$ as like Proof. A.2;
- For the invoking Π_{Pos} with verifier P_1:
  - $\mathcal{S}$ generate $\Delta^{(i)}$ with seed $\eta_{0,2}$;
  - $\mathcal{S}$ generate $x_1^{(i)} := [r_x^{(i)}]_2$ with seed $\eta_{0,2}$;
  - $\mathcal{S}$ generate random $\|r_k^{(i)}\|$ for $k \in \mathbb{Z}_\ell$ with random MAC key α and acts as verifier P_1 to share $\|r_k^{(i)}\|$ and α to P_0.
  - Upon receiving $\|u_k^{(i)}\|_0$ for $k \in \mathbb{Z}_\ell$ form P_0, $\mathcal{S}$ reconstructs $x_1'^{(i)}$ and its MACs share $\gamma(x_1'^{(i)})$.
  - If $x_1'^{(i)} \cdot \alpha \neq \gamma(x_1'^{(i)})$ or $x_1'^{(i)} \neq x_1^{(i)}$, $\mathcal{S}$ abort.
- Similarly, for the invoking Π_{Pos} with verifier P_2, $\mathcal{S}$ generates $\Delta'^{(i)}$ with seed $\eta_{0,1}$, generates $x_2^{(i)}$ from $[r_x^{(i)}]_1$, reconstructs $x_2'^{(i)}$ and its MACs share $\gamma(x_2'^{(i)})$. $\mathcal{S}$ aborts if $x_2'^{(i)} \cdot \alpha \neq \gamma(x_2'^{(i)})$ or $x_2'^{(i)} \neq x_2^{(i)}$.
- $\mathcal{S}$ sends $(\mathsf{Input}, \mathsf{sid}, (x_1^{(i)}, x_2^{(i)}))$ to $\mathcal{F}_{\mathsf{VSignBit}}$;
- $\mathcal{S}$ calculates $\delta^{(i)} = r_x'^{(i)} - x_1^{(i)} - x_2^{(i)}$;
- For the invoking $\Pi_{\mathsf{MultVerify}}$,
  - $\mathcal{S}$ picks t_1 and t_2 and share them to P_0.
  - If $\delta_1^{(i)} \neq \delta_2^{(i)}$, $\mathcal{S}$ reveal $\beta \neq 0$ to P_0 as result of $\Pi_{\mathsf{MultVerify}}$.
  - $\mathcal{S}$ extract $t_0' = t_1 - t_0, t_1' = t_2 - t_0$ from the execution of $\Pi_{\mathsf{MultVerify}}$.
  - If $t_1 - t_0' \neq t_2 - t_1'$, $\mathcal{S}$ reveal $\beta \neq 0$ to P_0 as result of $\Pi_{\mathsf{MultVerify}}$.
  - if $t_0 \neq H(\Delta^{(0)}||\ldots||\Delta^{(N-1)}||\Delta'^{(0)}||\ldots||\Delta'^{(N-1)})$, $\mathcal{S}$ reveal $\beta \neq 0$ to P_0 as result of $\Pi_{\mathsf{MultVerify}}$.
  - $\mathcal{S}$ sets $\mathsf{Alg}(\left\{(r_1^{(i)}, r_2^{(i)}, m_1^{(i)}, m_2^{(i)})\right\}_{i\in\mathbb{Z}_N}) := \{\mathsf{sign}(m_1^{(i)} - r_1^{(i)} - r_2^{(i)} - \delta^{(i)})\}_{i\in\mathbb{Z}_N}$;
  - $\mathcal{S}$ sets $f(\left\{(r_1^{(i)}, r_2^{(i)}, m_1^{(i)}, m_2^{(i)})\right\}_{i\in\mathbb{Z}_N}) := \bigwedge_{i=0}^{i=N-1}(\mathsf{sign}(m_1^{(i)} - r_1^{(i)} - r_2^{(i)} - \delta^{(i)}) + \delta_1^{(i)} \stackrel{?}{=} \mathsf{sign}(m_1^{(i)} - r_1^{(i)} - r_2^{(i)}))$;
  - $\mathcal{S}$ sends $(\mathsf{Compute}, \mathsf{sid}, \mathsf{Alg}, f)$ to $\mathcal{F}_{\mathsf{VSignBit}}$;
  - Upon receiving $(\mathsf{Leak}, \mathsf{sid}, b)$ from $\mathcal{F}_{\mathsf{VSignBit}}$, reveals random value $\beta \neq 0$ to P_0 if $b = 0$, else reveals $\beta = 0$ to the corrupted P_0.

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 $\mathsf{Real}_{\Pi_{\mathsf{VSignBit}},\mathcal{A},\mathcal{Z}}(1^\kappa)$.

Hybrid $\mathcal{H}_1$: It is same as $\mathcal{H}_0$ except that in $\mathcal{H}_1$, list $\|r_k^{(i)}\|$ of Π_{Pos} is picked uniformly random.

Hybrid $\mathcal{H}_2$: It is the idea world execution $\mathsf{Ideal}_{\mathcal{F}_{\mathsf{VSignBit}},\mathcal{S},\mathcal{Z}}(1^\kappa)$ which is same as $\mathcal{H}_1$ except that in $\mathcal{H}_2$, t_1, t_2 and β is picked random.

Claim 3. *If* $\mathsf{PRF}^{\mathbb{Z}_p}$ *and* $\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}$ *are the secure pseudorandom functions with adversarial advantage* $\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_p}}(1^\kappa,\mathcal{A})$ *and advantage* $\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}}(1^\kappa,\mathcal{A})$, *then* $\mathcal{H}_1$ *and* $\mathcal{H}_0$ *are indistinguishable with advantage* $2\cdot\ell\cdot\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_p}}(1^\kappa,\mathcal{A}) + 3\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}^\ell}}(1^\kappa,\mathcal{A})$.

Proof. For $\|r_k^{(i)}\|$ of $\mathcal{H}_0$ and $\mathcal{H}_1$, we replace 2ℓ $\mathsf{PRF}^{\mathbb{Z}_p}$ outputs to uniformaly random. For t_1 and t_2 of $\mathcal{H}_1$ and $\mathcal{H}_2$, we replace 2 $\mathsf{PRF}^{\mathbb{Z}_\ell}$ outputs to uniformaly random. For β, we replace PRF^{2^ℓ} outputs to uniformaly random.Therefor, the overall advantage is $\epsilon = 2\cdot\ell\cdot\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_p}}(1^\kappa,\mathcal{A}) + 3\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}^{2^\ell}}}(1^\kappa,\mathcal{A})$

□

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_1, P_2. $\mathcal{S}$ simulates random oracle RO and the following interactions with $\mathcal{A}$.

- $\mathcal{S}$ generates $[r_x^{(i)}]_1$ with seed $\eta_{0,1}$.
- $\mathcal{S}$ extracts $m_x'^{(i)}$ and $\delta^{(i)}$as like Proof. A.2
- $\mathcal{S}$ sends $(\mathsf{input}, \mathsf{sid}, m_x'^{(i)}, [r_x^{(i)}]_1)$ to $\mathcal{F}_{\mathsf{VSignBit}}$.
- For the invoking Π_{Pos} with verifier P_2:
  - $\mathcal{S}$ generate $\Delta^{(i)}$ with seed $\eta_{1,2}$.
  - $\mathcal{S}$ generate random $\|r_k^{(i)}\|$ for $k \in \mathbb{Z}_\ell$ with random MAC key α and acts as verifier P_2 to share $\|r_k^{(i)}\|$ and α to P_1.
  - Upon receiving $\|u_k^{(i)}\|_0$ for $k \in \mathbb{Z}_{\ell+1}^*$ form P_1, $\mathcal{S}$ reconstructs $x_2'^{(i)}$ and its MACs share $\gamma(x_2'^{(i)})$.
  - If $x_2'^{(i)}\cdot\alpha \neq \gamma(x_2^{(i)})$ or $[r_x^{(i)}]_1 \neq x_2'^{(i)}$, $\mathcal{S}$ abort.
- For the invoking Π_{Pos} with verifier P_1.
  - extracts $r^{(i)}$ from the message received from P_1.
  - simulates the list (u_k,u_k') like Proof. A.2.
  - calculates $v_1'^{(i)}$ form $\mathsf{sign}(r)$ and the list (u_k,u_k');
  - calculates $m_x''^{(i)} = r^{(i)} - [r_x^{(i)}]_1$.
- For the invoking $\Pi_{\mathsf{MultVerify}}$,
  - $\mathcal{S}$ picks t_0 and t_2 and share them to P_0.
  - $\mathcal{S}$ extracts $t_0' = t_2 - t_1$, $t_1' = t_1 - t_0$ from the execution of $\Pi_{\mathsf{MultVerify}}$.
  - If $t_2 - t_0' \neq t_1 - t_1'$, $\mathcal{S}$ reveal $\beta \neq 0$ to P_0 as result of $\Pi_{\mathsf{MultVerify}}$.
  - $\mathcal{S}$ sets $t_1 = t_2 - t_0'$.
  - $\mathcal{S}$ sets $\mathsf{Alg}(\left\{(r_1^{(i)}, r_2^{(i)}, m_1^{(i)}, m_2^{(i)})\right\}_{i\in\mathbb{Z}_N}) := \{\mathsf{Alg}(r_1^{(i)}, r_2^{(i)}, m_1^{(i)}, m_2^{(i)}) + \delta^{(i)}\}_{i\in\mathbb{Z}_N}$, where Alg is same construnction in Proof. A.2;
  - $\mathcal{S}$ sets $f(\left\{(r_1^{(i)}, r_2^{(i)}, m_1^{(i)}, m_2^{(i)})\right\}_{i\in\mathbb{Z}_N})$ as
    - return 0 if exists $\mathsf{Alg}(r_1^{(i)}, r_2^{(i)}, m_1^{(i)}, m_2^{(i)}) + \delta^{(i)} \notin \{0,1\}$.
    - $f_0 := \Pi_{i=0}^{N-1}(\mathsf{Alg}(r_1^{(i)}, r_2^{(i)}, m_1^{(i)}, m_2^{(i)}) + \delta^{(i)} \overset{?}{=} \mathsf{sign}(m_2^{(i)} - r_1^{(i)} - r_2^{(i)}))$;
    - $v_1''^{(i)} = v_1'^{(i)}$ if $\mathsf{Alg}(r_1^{(i)}, r_2^{(i)}, m_1^{(i)}, m_2^{(i)}) + \delta^{(i)} \overset{?}{=} \mathsf{sign}(m_x''^{(i)} - r_1^{(i)} - r_2^{(i)})$. $v_1''^{(i)} = v_1'^{(i)} \oplus 1$ if $\mathsf{Alg}(r_1^{(i)}, r_2^{(i)}, m_1^{(i)}, m_2^{(i)}) + \delta^{(i)} \overset{?}{\neq} \mathsf{sign}(m_x''^{(i)} - r_1^{(i)} - r_2^{(i)})$ for $i \in \mathbb{Z}_N$.
    - $f := f_0\cdot(H(\Delta^{(0)}||\ldots||\Delta^{(N-1)}||v_1''^{(0)}||\ldots||v_1''^{(N-1)}) \overset{?}{=} t_1)$;
  - $\mathcal{S}$ sends $(\mathsf{Compute}, \mathsf{sid}, \mathsf{Alg}', f)$ to $\mathcal{F}_{\mathsf{VSignBit}}$;
  - Upon receiving $(\mathsf{Leak}, \mathsf{sid}, b)$ from $\mathcal{F}_{\mathsf{VSignBit}}$, reveals random value $\beta \neq 0$ to P_0 if $b = 0$, else reveals $\beta = 0$ to the corrupted P_0.

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 $\mathsf{Real}_{\Pi_{\mathsf{VSignBit}},\mathcal{A},\mathcal{Z}}(1^\kappa)$.

Hybrid $\mathcal{H}_1$: It is same as $\mathcal{H}_0$ except that in $\mathcal{H}_1$, list t_0 and t_1 are picked random instead of calculated with H.

Hybrid $\mathcal{H}_2$: It is the idea world execution $\mathsf{Ideal}_{\mathcal{F}_{\mathsf{VSignBit}},\mathcal{S},\mathcal{Z}}(1^\kappa)$ which is same as $\mathcal{H}_1$ except that in $\mathcal{H}_2$, β is picked random instead of calculated with $\Pi_{\mathsf{MultVerify}}$.

Claim 4. *If* $\mathsf{PRF}^{\mathbb{Z}_p}$ *and* $\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}$ *are the secure pseudorandom functions with adversarial advantage* $\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_p}}(1^\kappa,\mathcal{A})$ *and* $\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}}(1^\kappa,\mathcal{A})$, *then* $\mathcal{H}_2$ *and* $\mathcal{H}_0$ *are indistinguishable with advantage* $\epsilon = \ell\cdot\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_p}}(1^\kappa,\mathcal{A}) + 3\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}}(1^\kappa,\mathcal{A})$.

□

Appendix B.
Other Aegis component

B.1. Inner product and convolution.

Our maliciously secure inner product protocol Π_{Inner} is shown in Fig. 19. 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 the malicious setting, we employ batch verification protocol $\Pi_{\mathsf{InnerVerify}}^{R}$ (Fig. 18) to ensure the correctness of the inner products with a similar manner of multiplication. Analogously, in $\Pi_{\mathsf{InnerVerify}}^{R}$, 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

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 18: The Batch Inner Product Verification Protocol

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 19: The Inner Product Protocol

m separate inner products. We implement these two types of operations by invoking Π_{Inner} a total of m times.

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

Let $\mathsf{rshift}(x, y)$ denote right shift x with y bits. 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 20: The maliciously secure truncation protocol

B.2. 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 [27], [30], [33] 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. 20, we also design a maliciously secure probabilistic truncation protocol Π^{t}_{Trunc} for the truncation bit size t. Our idea is similar to SWIFT [27] which generates correct truncation pair via maliciously secure inner product protocol. However, in contrast to SWIFT [27], 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 [27] 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 share 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 Fig. 21,

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

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

Preprocessing:

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

Online:

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

Figure 21: The semi-honest truncation protocol

TABLE 3: Run-time and communication cost of NN inference, under LAN setting with batch size 30. (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

which only requires one round and communication of ℓ bits in the offline phase.

B.3. ReLU

Our 2-round ReLU protocol is depicted in Fig. 22.

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

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

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 $\hat{r}_x = -r_x - \textsf{sign}(-r_x) \cdot 2^{\ell-1} \in \mathbb{Z}_{2^\ell}$;
  2) extract $2^{\ell-1} - 1 - \hat{r_x}$ as $\{r_{x,0}, \ldots, r_{x,\ell-2}\}$;
  3) perform $\llbracket r_{x,i} \rrbracket^p \leftarrow \Pi^p_{\llbracket\cdot\rrbracket}(r_{x,i})$ for $i \in \mathbb{Z}_{\ell-1}$, 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_j, for $j \in \{1, 2\}$ does:
  1) set $\hat{m}_x = m_x - \textsf{sign}(m_x) \cdot 2^{\ell-1}$ and bitexact it as $\{\hat{m}_{x,i} \in \{0, 1\}\}_{i \in \mathbb{Z}_\ell}$ while $\sum_{i=0}^{\ell-1} 2^{\ell-1-i} \hat{m}_{x,i} = \hat{m}_x$;
  2) set $\hat{m}_{x|\ell} = 0$ and $\llbracket r_{x,\ell} \rrbracket = \llbracket 1 \rrbracket$;
  3) set $\llbracket m_i \rrbracket^p = \hat{m}_{x,i} + \llbracket r_{x,j} \rrbracket^p - 2\hat{m}_{x,i} \cdot \llbracket r_{x,i} \rrbracket^p$ for $i \in \mathbb{Z}_\ell$.
  4) pick same random values $\{w_i, w'_i \in \mathbb{Z}^*_p\}_{i \in \mathbb{Z}_\ell}$ via PRF with seed $\eta_{1,2}$;
  5) calculate $\llbracket m'_i \rrbracket^p = \sum_{t=1}^{i} \llbracket m_t \rrbracket^p - 2 \cdot \llbracket m_i \rrbracket^p + 1$ and $\llbracket u_i \rrbracket^p = w_i \cdot \llbracket m'_i \rrbracket^p + (\textsf{sign}(m_x) \oplus \hat{m}_{x,i} \oplus \Delta)$ and $\llbracket u'_i \rrbracket^p = w'_i(w_i \cdot \llbracket m'_i \rrbracket^p + 1)$ for $i \in \mathbb{Z}_\ell$;
  6) pick a random permutation π via PRF with seed $\eta_{1,2}$ and permute the list $\{\llbracket \hat{u}_i \rrbracket^p\}_{i \in \mathbb{Z}_\ell} = \pi(\{\llbracket u_i \rrbracket^p\}_{i \in \mathbb{Z}_\ell})$ and $\{\llbracket \hat{u}'_i \rrbracket^p\}_{i \in \mathbb{Z}_\ell} = \pi(\{\llbracket u'_i \rrbracket^p\}_{i \in \mathbb{Z}_\ell})$;
  7) reveal $\{\llbracket \hat{u}_i \rrbracket^p\}_{i \in \mathbb{Z}_\ell}$ and $\{\llbracket \hat{u}'_i \rrbracket^p\}_{i \in \mathbb{Z}_\ell}$ to P_0 and reveal $\Gamma'' = m_x \cdot [r_z] + [\Gamma']$ to each other simultaneously;
- P_0 sets $m' = \textsf{sign}(-r_x) - r'$ if $\exists \hat{u}_i = 0 \wedge \hat{u}'_i \neq 0$ for $i \in \mathbb{Z}_\ell$ else $m' = (1 \oplus \textsf{sign}(-r_x)) - r'$;
- P_0 sets $m'' = m' \cdot r_x + r''$;
- P_0 sends m' and m'' to P_j, for $j \in \{1, 2\}$;
- P_j, for $j \in \{1, 2\}$ 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 22: The 2-round ReLU Protocol.

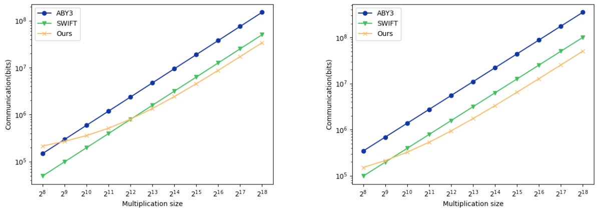
Appendix C.
Benchmarks

C.1. The inference of neural network.

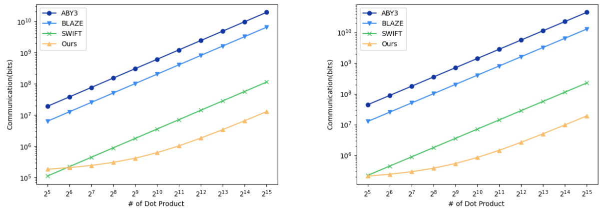
We further construct the convolutional neural network (CNN) inference. We implement three types of models as follows:

- 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 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.



(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 23: Communication overhead comparison with ABY3 [30], BLAZE [33], SWIFT [27] of muliplication and inner product.

C.2. Multiplication communication comparison

Fig. 23 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 $(6R + 5)\ell \cdot d$ (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.3. Non-arithmetic protocol benchmark in semi-honest setting

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

C.4. Performance comparisons of P-Falcon [38] and our ReLU protocols

TABLE 5 shows the performance comparison between our semi-honest ReLU protocol and Falcon under piranha code [40]. Our protocol achieves a performance improvement of more than $3\times$ compared to Falcon [38].

C.5. 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 6.

Appendix D.
Related work

In the honest-majority setting, several works such as [13], [16], [18], [19], [21], [42] 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, [42] 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. [13] reduces the communication overhead to $42 \cdot n$. [21] introduces batch verification and a series of other optimization techniques. These protocols by [21] 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 [21]'s protocol can only run on the field. In contrast, [19] achieves a constant online phase communication overhead of 12 field elements by utilizing packed secret sharing technology. Lastly, the work by [16] 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. [8], [10], [10], [11], [27], [30], [33], [34], [38] realize privacy-preserving machine learning protocols under the malicious threat model in an honest majority. In the semi-honest setting, protocols such as [10], [30], [32], [33] 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 [10]. In the malicious setting, different from the overhead of 21 ring elements (12 in the offline phase) [30], a series of optimizations [10], [27], [33] 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 [27], [30], [32] 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 of communication. Furthermore, in protocols such as [10], [30], [33], 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 [28], [37] 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,

TABLE 4: Runtime and communication cost of each non-arithmetic protocol evaluation in semi-honest, MAN setting.(ops) for operations per second.

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: Performance comparisons of P-Falcon [38], [40] and our ReLU protocols on the different networks and batch sizes. (ops) for operations per second.

Batch	**Protocol**	**LAN**		**MAN**	
		Time	**Thr. (ops)**	**Time**	**Thr. (ops)**
2^{10}	P-Falcon [38], [40]	9914.1μs	93541.87	313616μs	3188.61
	Ours	4160.3μs	240367.28	93391.5μs	10707.61
2^{14}	P-Falcon [38], [40]	22128.5μs	451905.91	452435μs	22102.62
	Ours	8313.8μs	1202819.4	99684.4μs	100316.59
2^{18}	P-Falcon [38], [40]	152434μs	656021.62	2171200μs	46057.48
	Ours	47193.1μs	2118953.83	397612.3μs	251501.27

TABLE 6: 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	$R+1$	$(6R+3N/2^R+6)\ell\cdot d$
Inner Product	ℓ	1	2ℓ	$R+1$	$(6R+3N\cdot M/2^R+6)\ell\cdot d$
Truncation	ℓ	0	0	$R+1$	$(6R+6N/2^R+6)\ell\cdot d$
Sign-bit Extraction	$(\ell-1)\log\ell)+2\ell$	2	$4\ell\log\ell+2\ell$	2	$2((\lambda+1)(\ell-1)\log\ell+6\ell\log\ell+\ell)$
ReLU	$(\ell-1)\log\ell)+4\ell$	2	$4\ell\log\ell+4\ell$	2	$2((\lambda+1)(\ell-1)\log\ell+6\ell\log\ell+2\ell)$
MaxPool	$(n-1)(4\ell+\ell\log\ell)$	$\log n$	$(n-1)4\ell(\log\ell+2)$	$2\log n$	$2(n-1)((\lambda+1)(\ell-1)\log\ell+6\ell\log\ell+\ell)$

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, [43] 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.