

The Communication-Friendly Privacy-Preserving Machine Learning against Malicious Adversaries

Tianpei Lu, Bingsheng Zhang, *Member, IEEE*, Lichun Li, and Kui Ren, *Fellow, IEEE*

Abstract—With the increasing emphasis on privacy regulations, such as GDPR, protecting individual privacy and ensuring compliance have become critical concerns for both individuals and organizations. Privacy-preserving machine learning (PPML) is an innovative approach that allows for secure data analysis while safeguarding sensitive information. It enables organizations to extract valuable insights from data without compromising privacy. Secure multi-party computation (MPC) is a key tool in PPML, as it allows multiple parties to jointly compute functions without revealing their private inputs, making it essential in multi-server environments. We address the performance overhead of existing maliciously secure protocols, particularly in finite rings like \mathbb{Z}_{2^ℓ} , by introducing an efficient protocol for secure linear function evaluation. We implement our maliciously secure MPC protocol on GPUs, significantly improving its efficiency and scalability. We extend the protocol to handle linear and non-linear layers, ensuring compatibility with a wide range of machine-learning models. Finally, we comprehensively evaluate machine learning models by integrating our protocol into the workflow, enabling secure and efficient inference across simple and complex models, such as convolutional neural networks (CNNs).

I. INTRODUCTION

In the era of big data, privacy protection and compliance have become paramount concerns for both individuals and organizations. As various privacy regulations, such as GDPR, have emerged, the demand for effective privacy-preserving mechanisms has intensified significantly. Privacy-preserving machine learning (PPML) is an innovative technique that enhances privacy while enabling secure data mining and machine learning. It ensures that sensitive information remains confidential, allowing organizations to leverage data insights without compromising individual privacy.

Secure multi-party computation (MPC) [1]–[3] allows multiple parties to jointly evaluate functions without revealing their private inputs. This cryptographic tool plays a crucial role in realizing PPML in multi-server environments [4]–[9]. Notably, this work focuses on 3-party MPC, referred to as 3-PC. Most existing protocols [10], [11] are designed for a semi-honest setting, where participants are assumed to adhere to the protocol and act honestly, albeit with the potential to glean additional information from the data they handle. However, in many scenarios, the importance of robust defenses against malicious actors becomes critical. Maliciously secure protocols are essential in these contexts, as they can detect adversarial behaviors and protect the integrity of the computation.

Despite the advancements, state-of-the-art maliciously secure PPML protocols face significant performance overhead. For instance, maliciously secure multiplication protocols can be at least twice as slow as their semi-honest counterparts [12], [13]. This performance gap raises concerns, especially given that PPML-friendly MPC protocols typically operate over finite rings like \mathbb{Z}_{2^ℓ} , which facilitate fixed-point arithmetic. Designing maliciously secure MPC over \mathbb{Z}_{2^ℓ} is inherently more complex than over prime-order finite fields \mathbb{Z}_p .

Recently, several works [14]–[16] have successfully implemented efficient maliciously secure protocols over \mathbb{Z}_p . However, techniques used to achieve malicious security in \mathbb{Z}_p cannot be directly applied to \mathbb{Z}_{2^ℓ} due to the absence of inverses for certain elements. Attempts to adapt these techniques have resulted in protocols that incur a twofold communication overhead. Alternatively, some research efforts [5], [12], [13] aim to develop maliciously secure MPC over \mathbb{Z}_{2^ℓ} from the ground up. Nonetheless, these solutions often generate significantly higher communication overhead compared to semi-honest protocols. This performance loss is particularly troubling in today's economic landscape, where communication costs on platforms like Amazon can far surpass computation costs, underscoring the urgent need for efficient, secure protocols that balance both privacy and performance.

Our results. In this work, we improve the performance of maliciously secure linear functions evaluation for enhanced PPML. Our protocols are 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) [11]; 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 $r = r_1 + r_2$.

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.* [14] 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.* [14] 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 [17]–[19] to adopt the multiplication verification over the field to the ring are not suitable for the protocol proposed in [14]. To resolve the soundness issue, we extend the shared elements over \mathbb{Z}_{2^ℓ} to the quotient ring of polynomials $\mathbb{Z}_{2^\ell}[x]/f(x)$ [20]–[22], where $f(x)$ is a degree- d irreducible polynomial over \mathbb{Z}_{2^ℓ} to

T. Lu, B. Zhang and K. Ren are with The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Zhejiang University, Hangzhou, China. E-mail: {lutianpei, bingsheng, kuiren}@zju.edu.cn. L. Li is with Ant group, Hangzhou, China. E-mail: lichun.llc@antgroup.com. B. Zhang is Corresponding Author.

apply the Lagrange interpolating based dimension reduction technique [14]. Consequently, the overall communication of our batch multiplication verification protocol is logarithmic to the number of multiplication gates.

Our protocols are compatible with mixed-circuit computation. Previous research [4], [23]–[25] has shown that computing non-linear functions, such as comparison, is more efficient in binary computation. This necessitates switching between arithmetic and binary computation, as arithmetic is superior for dot products. Rotaru and Wood introduced the concept of double-authenticated bits (daBits) [26], which are secret random bits shared across both arithmetic and binary. We observe that our protocol can be directly applied to daBits with minimal modifications. By utilizing daBits, we enable secure evaluation of any non-linear function under malicious security.

Finally, we integrated both linear and non-linear functions to systematically evaluate machine learning models.

Performance. Table I depicts the comparison between our protocols and SOTA 3PC maliciously secure protocol. As we can see, our protocols achieve a significant communication reduction.

Batch verification for multiplication over the ring.

Compared with the prime-order finite field, constructing an MPC over ring \mathbb{Z}_{2^ℓ} 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^ℓ} 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.* [13], our protocol reduces the overall communication by 40%. Note that Dalskov *et al.* [13] achieves full security in the \mathcal{Q}^3 active adversary setting ($t < n/3$), while our protocol achieves security with abort in the \mathcal{Q}^2 active adversary setting ($t < n/2$), where t is the number of corrupted parties and n is the total number of participants. Compared with SOTA 3PC multiplication over ring [12], 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 [12].

Implementation with GPUs. Since our implementation requires converting secret sharing to an extended ring during the verification phase, this introduces significant computational overhead. However, the extended ring offers excellent concurrency, allowing us to implement our protocol on GPUs. In our specific experiments, compared to ABY3, our implementation achieved a threefold performance improvement, and when compared to Swift, we realized a twofold increase in performance.

Implementation of maliciously secure PPML framework.

We built a comprehensive privacy-preserving machine learning application against malicious adversaries based on Piranha [27] framework. This includes the implementation

of typical CNN models such as VGG and ResNet. Our framework delineates between semi-honest offline and online computation phases, as well as a separate multiplication gate (for both arithmetic and boolean) verification phase. Our experiments demonstrate that the time overhead of the verification phase is significantly lower than that of the online computation phase, indicating that the time introduced by malicious security is far less than the original cost of the semi-honest protocol.

Paper Organization. We first propose our maliciously secure 3PC in Sec. III. In Sec. IV, we realize the PPML framework based on our maliciously secure protocols for both linear and non-linear operation. In Sec. V, we benchmark the performance of our protocols and PPML framework.

II. 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 [4], [5]: 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 $2^\ell + \lfloor x \cdot 2^k \rfloor$. We use $\eta_{j,k}$ to denote the common seed held by P_j and P_k . Our protocol contains two types of secret sharing as follows:

- $[\cdot]^\ell$ -sharing: We define $[\cdot]^\ell$ -sharing over ring \mathbb{Z}_{2^ℓ} as $[x]^\ell := ([x]_1 \in \mathbb{Z}_{2^\ell}, [x]_2 \in \mathbb{Z}_{2^\ell})$ where $x = [x]_1^\ell + [x]_2^\ell$. P_j for $j \in \{1, 2\}$ hold share $[x]_j^\ell$.
- $\langle \cdot \rangle^\ell$ -sharing: We define $\langle \cdot \rangle^\ell$ -sharing over ring \mathbb{Z}_{2^ℓ} as $\langle x \rangle^\ell := ([r_x]^\ell, m_x)$ where r_x is a fresh random value and $m_x = r_x + x$. P_j for $j \in \{1, 2\}$ hold $(m_x \in \mathbb{Z}_{2^\ell}, [r_x]_j^\ell \in \mathbb{Z}_{2^\ell})$ and P_0 holds $([r_x]_1^\ell, [r_x]_2^\ell)$.

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 simplicity, we use $[\cdot]$, $\langle \cdot \rangle$ 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 $\langle \cdot \rangle$, and $\langle \cdot \rangle^{\mathbb{Z}_{2^\ell}[x]}$.

Secret sharing. Let $\Pi_{[\cdot]}$ and $\Pi_{\langle \cdot \rangle}$ 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 [28]. 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.

- $[x]^\ell \leftarrow \Pi_{[\cdot]}^\ell(x)$: (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 .
- $[x]^\ell \leftarrow \Pi_{[\cdot]}^\ell$: (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 $\eta_{0,2}$;

TABLE I: Comparison of 3-PC based PPML. (ℓ is the ring size, n is the size of the inner product.)

Operation	Protocol	Offline Communication (bits)	Online		Malicious
			Rounds	Communication (bits)	
Mult	ABY3 [4]	12ℓ	1	9ℓ	✓
	BLAZE [5]	3ℓ	1	3ℓ	✓
	SWIFT [12]	3ℓ	1	3ℓ	✓
	Ours	1ℓ	1	2ℓ	✓
Inner Product	ABY3 [4]	$12n\ell$	1	$9n\ell$	✓
	BLAZE [5]	$3n\ell$	1	3ℓ	✓
	SWIFT [12]	3ℓ	1	3ℓ	✓
	Ours	1ℓ	1	2ℓ	✓
Inner Product with Truncation	ABY3 [4]	$12n\ell + 84\ell$	1	$9n\ell + 3\ell$	✓
	BLAZE [5]	$3n\ell + 2\ell$	1	3ℓ	✓
	SWIFT [12]	15ℓ	1	3ℓ	✓
	Ours	7ℓ	1	2ℓ	✓

- P_0 calculates $x = [x]_1 + [x]_2$.
- $\langle x \rangle^\ell \leftarrow \Pi_{\langle \cdot \rangle}^{\ell, k}(x)$: (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 .
- $\langle x \rangle^\ell \leftarrow \Pi_{\langle \cdot \rangle}^\ell$: (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 $\eta_{1,2}$.

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

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 shares will be detected by the honest parties.

Formally, the verifiable reconstruct protocol Π_{Rec} is described as follows:

- $x \leftarrow \Pi_{\text{Rec}}(\langle x \rangle)$:
 - 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 $x = m_x - [r_x]_1 - [r_x]_2$.
- $x \leftarrow \Pi_{\text{Rec}}^{\ell, k}(\langle x \rangle)$: 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 $x = m_x - [r_x]_1 - [r_x]_2$.

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

Preprocessing and postprocessing. We follow the “preprocessing” paradigm [29], 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 [19], where batch verification is performed.

Multiplication gate. We adopt the multiplication protocol of ASTRA [11]. 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

$$m_z = xy + r_z = (m_x - r_x)(m_y - r_y) + r_z$$

$$= \underbrace{m_x m_y - m_x r_y - m_y r_x}_{P_1 \text{ and } P_2 \text{ can locally evaluate}} + \underbrace{r_x r_y + r_z}_{\text{Known to } P_0}.$$

$[\Gamma'] = m_x m_y - m_x [r_y] - m_y [r_x]$ can be calculated by P_1 and P_2 locally and $[\Gamma] = [r_x \cdot r_y] - [r_z]$ can be secret shared by P_0 to P_1 and P_2 in the preprocessing phase. In the online phase, P_1 and P_2 calculate and reconstruct $[m_z] = [\Gamma'] + [\Gamma]$.

Inner product. Given an arbitrary dimension inner product, its communication cost equals to a single multiplication. Considering n -dimension inner product $z = \sum_{i=0}^{n-1} x_i \cdot y_i$, the artifact m_z requires to be evaluated in online phase can be written as

$$m_z = \sum_{i=0}^{n-1} x_i \cdot y_i + r_z = \sum_{i=0}^{n-1} (m_{x_i} - r_{x_i})(m_{y_i} - r_{y_i}) + r_z$$

$$= \underbrace{\sum_{i=0}^{n-1} (m_{x_i} m_{y_i} - m_{x_i} r_{y_i} - m_{y_i} r_{x_i})}_{P_1 \text{ and } P_2 \text{ can locally evaluate}} + \underbrace{\sum_{i=0}^{n-1} r_{x_i} r_{y_i} + r_z}_{\text{Known to } P_0}.$$

Similar to single multiplication, $[\Gamma'] = \sum_{i=0}^{n-1} (m_{x_i} m_{y_i} - m_{x_i} [r_{y_i}] - m_{y_i} [r_{x_i}])$ can be locally evaluated by P_1 and P_2 . Meanwhile, $[\Gamma] = \sum_{i=0}^{n-1} [r_{x_i} r_{y_i}] + [r_z]$ can be secret shared by P_0 to P_1 and P_2 in the offline phase. In the online phase, P_1 and P_2 compute $[m_z] = [\Gamma] + [\Gamma']$ and reconstruct m_z .

Security up to additive attacks. A protocol is secure up to additive attacks when all behaviors the adversary performs can only introduce an additive error known to the adversary to the output of the protocol. As proven in [30], the typical replicated secret sharing protocol, such as aforementioned multiplication and inner product, 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. Our protocol and framework achieve active security with abort in an honest majority setting, while one arbitrary party in \mathcal{P} is under the control of a static malicious adversary. We emphasize abort security with computational soundness, ensuring that malicious behavior will be detected with overwhelming probability.

III. 3PC WITH MALICIOUS SECURITY

We use the postprocessing verification procedure to detect any potential malicious behavior. Before reconstructing the final result, an extra verification is performed to ensure the correctness of the final result. Our maliciously secure protocol is based on the additive security of RSS, namely, the corresponding protocol is secure up to additive attacks.

Correctness Verification for Arithmetic Circuit. For a circuit containing both multiplication and addition gates, the correctness verification of the overall circuit using 3PC replicated shares reduces to verifying all multiplication gates. When an adversary introduces an error at an addition gate, since addition is non-interactive, it will cause an inconsistency in the shares. As previously mentioned, replicated shares possess a verifiable reconstruction property against a single malicious party. In the multiplication operation $z = x \cdot y$, P_0 can introduce an error when sharing $[r_x \cdot r_y]$, while P_1 and P_2 can introduce errors during the reconstruction of m_z , without breaking share consistency. Denoting the set of multiplication gates by \mathcal{G} , the verification checks the following equation:

$$\bigwedge_{\{x^{(i)}, y^{(i)}, z^{(i)}\} \in \mathcal{G}} x^{(i)} \cdot y^{(i)} = z^{(i)} \quad (1)$$

To batch verify multiple multiplication gates $\langle x^{(i)} \rangle, \langle y^{(i)} \rangle, \langle z^{(i)} \rangle_{i \in |\mathcal{G}|}$, we verify that the following inner product equals zero:

$$\Delta = \sum_{i=0}^{|\mathcal{G}|} (r^i \cdot x^{(i)} \cdot y^{(i)} - r^i \cdot z^{(i)}) = 0 \quad (2)$$

where r is a challenge picked during verification. The terms r^i prevent an adversary from introducing opposing errors in different outputs z_i and z_j that could cancel each other. For example, if $z^{(i)} = x^{(i)} \cdot y^{(i)} + e$ and $z^{(j)} = x^{(j)} \cdot y^{(j)} - e$, then $z^{(i)} + z^{(j)} = x^{(i)} \cdot y^{(i)} + x^{(j)} \cdot y^{(j)}$, making the error undetectable.

However, directly evaluating the inner product poses challenges. One challenge is that the adversary, knowing the additive error in $\langle z^{(i)} \rangle$, could cancel out the error to fabricate $\Delta = 0$. A typical solution involves using a random factor α . Instead of the 2-degree inner product, verification becomes a 3-degree polynomial:

$$\Delta = \sum_{i=0}^{|\mathcal{G}|} (r^i \cdot \alpha \cdot x^{(i)} \cdot y^{(i)} - r^i \cdot \alpha \cdot z^{(i)}) = 0 \quad (3)$$

where α is a random share unknown to each party. This randomness α serves as an additional layer of security by making it difficult for a malicious adversary to manipulate the values of the inputs and outputs in a way that cancels out errors introduced during verification. If the evaluation of this 3-degree polynomial is secure against additive attacks, the adversary can only introduce an input-independent error e' in Δ . To cancel the original error e in $z^{(i)}$, the adversary must guess $e' = \alpha \cdot e$. Since α is unknown and chosen randomly, the probability of correctly guessing the exact value of $\alpha \cdot e$ is extremely low.

Ring-Specific Challenges. The second challenge comes from irreversible multiplication in the ring. In ring-based computations, particularly over modular arithmetic, certain errors can exploit the properties of the ring to bypass verification. For instance, an adversary could introduce a specific error e such that when multiplied by r^i , it results in zero within the ring, even though the error itself is non-zero. Such chosen e will be undetected in a high probability if a lot of values α meets $e \cdot \alpha = 0$. A typical attack could involve introducing an error $e = 2^{\ell-1}$, where ℓ is the bit length of the ring. If r is an even number, this error would result in $r^i \cdot (z^{(i)} + e) = r^i \cdot z^{(i)}$, passing verification with a probability of 1/2.

One common solution to this problem is to increase the size of the ring used for verification, ensuring that the probability of an error passing undetected becomes vanishingly small. For example, in a protocol like SPDZ2k [18], a larger ring size (e.g., $\ell = 100$) is used for 64-bit data, resulting in a soundness error of 2^{-36} . In this scenario, even if the adversary tries to exploit the properties of the ring to introduce errors, the larger modulus significantly reduces the probability of success. Since converting shares from $\mathbb{Z}_{2^{64}}$ to $\mathbb{Z}_{2^{100}}$ is expensive, it is better to perform the arithmetic directly in $\mathbb{Z}_{2^{100}}$ rather than during the verification phase, which doubles the overhead. For smaller data ranges (e.g., 1-bit values), this overhead ratio increases.

Our approach is different. We perform Δ over the extension ring $\mathbb{Z}_{2^\ell}[x]/f(x)$, where $f(x)$ is an irreducible polynomial of degree d over \mathbb{Z}_{2^ℓ} [20]. (The original share over \mathbb{Z}_{2^ℓ} becomes the free coefficient, with d random elements added to the other coefficients.) According to the Schwartz-Zippel Lemma, the probability that a $|\mathcal{G}|$ -degree non-zero polynomial $\Delta(r) = 0$ for a randomly chosen r is at most $\frac{2^{(\ell-1)d}|\mathcal{G}|+1}{2^{\ell d}} \approx \frac{|\mathcal{G}|}{2^d}$.

Compared to the larger ring size approach, the extension ring offers two advantages: (i) Since the share conversion to the extended ring is non-interactive, there are no modifications required during the circuit evaluation phase for the semi-honest version of the protocol. This avoids any additional communication costs typically incurred during the verification phase. (ii) The extension ring approach is compatible with the dimensionality reduction technique proposed by [14], which reduces the communication complexity from $\Theta(|\mathcal{G}|)$ to $\Theta(\log |\mathcal{G}|)$. This optimization further improves the efficiency of the protocol, especially when dealing with a large number of multiplication gates.

In summary, our protocol operates as follows. First, we use a semi-honest protocol to evaluate the arithmetic circuit (on the ring \mathbb{Z}_{2^ℓ}). We then transform all the multiplication gate triplets to the extended ring $\mathbb{Z}_{2^\ell}[x]/f(x)$ and reformulate their

Protocol $\Pi_{\text{Trans}}(\{\langle x^{(i)} \rangle, \langle y^{(i)} \rangle, \langle z^{(i)} \rangle\}_{i \in \mathbb{Z}_N})$

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 $\langle r \rangle^{\ell[x]} \leftarrow \Pi_{\langle \cdot \rangle}^{\ell[x]}$ 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 $\{\langle x'^{(i)} \rangle^{\ell[x]}, \langle y^{(i)} \rangle^{\ell[x]}\}_{i \in \mathbb{Z}_N}; \langle z \rangle^{\ell[x]}$.

Fig. 1: Compression of Multiplication Triples.

verification as an inner product. Next, we apply the dimension reduction method from [14] to reduce the $|\mathcal{G}|$ -dimensional inner product to $\frac{|\mathcal{G}|}{2^d}$ dimensions. Finally, we use an inner product verification protocol to check the inner product after dimension reduction.

Compression of multiplication triples. We first design a subprotocol, Π_{Trans} (Fig. 1), which converts $|\mathcal{G}|$ multiplication triples over the ring \mathbb{Z}_{2^ℓ} into an $|\mathcal{G}|$ -dimensional inner product over the polynomial ring $\mathbb{Z}_{2^\ell}[x]/f(x)$ for verification.

The transformation begins by locally converting the multiplication triples $\{\langle x^{(i)} \rangle, \langle y^{(i)} \rangle, \langle z^{(i)} \rangle\}_{i \in \mathbb{Z}_{|\mathcal{G}|}}$ to the polynomial ring equivalents $\{\langle x^{(i)} \rangle^{\ell[x]}, \langle y^{(i)} \rangle^{\ell[x]}, \langle z^{(i)} \rangle^{\ell[x]}\}_{i \in \mathbb{Z}_{|\mathcal{G}|}}$. In this step, the free coefficient of the shares in $\mathbb{Z}_{2^\ell}[x]/f(x)$ is set to the original shares, while the remaining coefficients are padded with zero shares.

Next, the parties collectively generate a random challenge $r \in \mathbb{Z}_{2^\ell}[x]/f(x)$ by invoking the subprotocol $\langle r \rangle^{\ell[x]} \leftarrow \Pi_{\langle \cdot \rangle}^{\ell[x]}$, followed by reconstructing r via Π_{Rec} (To ensure that r is unknown to each party before circuit evaluation). Each party then locally computes $\langle z \rangle^{\ell[x]} = \sum_{i=0}^{|\mathcal{G}|-1} r^i \cdot \langle z^{(i)} \rangle^{\ell[x]}$ and $\langle x'^{(i)} \rangle^{\ell[x]} = r^i \cdot \langle x^{(i)} \rangle^{\ell[x]}$ for all $i \in \mathbb{Z}_{|\mathcal{G}|}$.

Finally, the protocol returns the $|\mathcal{G}|$ -dimensional inner product tuple as $(\{\langle x'^{(i)} \rangle^{\ell[x]}, \langle y^{(i)} \rangle^{\ell[x]}\}_{i \in \mathbb{Z}_{|\mathcal{G}|}}, \langle z \rangle^{\ell[x]})$.

Lemma 1. Suppose protocol Π_{Trans} take $\{\langle x^{(i)} \rangle, \langle y^{(i)} \rangle, \langle z^{(i)} \rangle\}_{i \in \mathbb{Z}_{|\mathcal{G}|}}$ as input, and it outputs $\{\langle x'^{(i)} \rangle^{\ell[x]}, \langle y^{(i)} \rangle^{\ell[x]}\}_{i \in \mathbb{Z}_{|\mathcal{G}|}}; \langle z \rangle^{\ell[x]}$. The probability that the following two conditions hold is at most $\frac{|\mathcal{G}|}{2^d}$, where d is the degree of $f(x)$ w.r.t. $\mathbb{Z}_{2^\ell}[x]/f(x)$:

- $z = \sum_{i=0}^{|\mathcal{G}|-1} x'_i \cdot y_i$
- $\exists i \in \mathbb{Z}_{|\mathcal{G}|}$ s.t. $z_i \neq x_i \cdot y_i$

Proof. It is sufficient to demonstrate that r is uniformly random, assuming that the reconstruction protocol Π_{Rec} does not abort. The adversary's goal is to manipulate the verification by ensuring that the following equation holds:

$$\sum_{i=0}^{|\mathcal{G}|-1} r^i \cdot z^{(i)} = \sum_{i=0}^{|\mathcal{G}|-1} r^i \cdot x^{(i)} \cdot y^{(i)}$$

Protocol $\Pi_{\text{Reduce}}(\{\langle x^{(i)} \rangle^{\ell[x]}, \langle y^{(i)} \rangle^{\ell[x]}\}_{i \in \mathbb{Z}_{|\mathcal{G}|}}, \langle z \rangle^{\ell[x]})$

Input : $|\mathcal{G}|$ -dimension $\langle \cdot \rangle^{\ell[x]}$ -shared inner product.

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

Execution:

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

Fig. 2: The Inner Product Dimension Reduction Protocol

where $z^{(i)} = x^{(i)} \cdot y^{(i)} + e^{(i)}$ for each $i \in \mathbb{Z}_{|\mathcal{G}|}$, and $e_{i \in \mathbb{Z}_{|\mathcal{G}|}}$ represents the list of errors introduced by the adversary at each gate. This can be written as

$$\sum_{i=0}^{|\mathcal{G}|-1} r^i \cdot x^{(i)} \cdot y^{(i)} = \sum_{i=0}^{|\mathcal{G}|-1} r^i \cdot (x^{(i)} \cdot y^{(i)} + e^{(i)})$$

By simplifying, we get:

$$\sum_{i=0}^{|\mathcal{G}|-1} r^i \cdot x^{(i)} \cdot y^{(i)} = \sum_{i=0}^{|\mathcal{G}|-1} r^i \cdot x^{(i)} \cdot y^{(i)} + \sum_{i=0}^{|\mathcal{G}|-1} r^i \cdot e^{(i)}$$

To satisfy this equation, the adversary must ensure that the error terms cancel out, which would require:

$$\sum_{i=0}^{|\mathcal{G}|-1} r^i \cdot e^{(i)}$$

This means that the adversary needs to find a value of r that is a root of the polynomial:

$$f(x) = \sum_{i=0}^{|\mathcal{G}|-1} x^i \cdot e^{(i)}$$

Since this polynomial is of degree at most $|\mathcal{G}| - 1$, the number of possible roots that satisfy the equation is limited. Specifically, for a degree- $\{|\mathcal{G}| - 1\}$ polynomial over the ring $\mathbb{Z}_{2^\ell}[x]$, according to the Schwartz-Zippel Lemma, the number of potential roots is bounded by $2^{(\ell-1)d}(|\mathcal{G}| + 1)$.

Thus, the probability that a uniformly random r selected during the protocol coincidentally matches one of these roots is given by:

$$\frac{2^{(\ell-1)d}(|\mathcal{G}| + 1)}{2^{\ell d}} \approx \frac{|\mathcal{G}|}{2^d}$$

□

Protocol $\Pi_{\text{InnerVerify}}(\{\langle x^{(i)} \rangle^{\ell[x]}, \langle y^{(i)} \rangle^{\ell[x]} \}_{i \in \mathbb{Z}_{|\mathcal{G}|}}, \langle z \rangle^{\ell[x]})$

Input : A $|\mathcal{G}|$ -dimension $\langle \cdot \rangle^{\ell[x]}$ -shared inner product pair.

Output : $z \stackrel{?}{=} \sum_{i=1}^{|\mathcal{G}|} x^{(i)} \cdot y^{(i)}$.

Execution:

- All parties invoke $\langle \alpha \rangle^{\ell[x]} \leftarrow \Pi_{\langle \cdot \rangle}^{\ell[x]}$;
- All parties calculate $\langle x'^{(i)} \rangle^{\ell[x]} = \langle x^{(i)} \rangle^{\ell[x]} \cdot \langle \alpha \rangle^{\ell[x]}$;
- All parties calculate $\langle \Delta \rangle^{\ell[x]} = \sum_{i=1}^{|\mathcal{G}|} \langle x'^{(i)} \rangle^{\ell[x]} \cdot \langle y^{(i)} \rangle^{\ell[x]} - \langle \alpha \rangle^{\ell[x]} \cdot \langle z \rangle^{\ell[x]}$;
- All parties call $\Delta = \Pi_{\text{Rec}}^{\ell[x]}(\langle \Delta \rangle^{\ell[x]})$;
- All parties output 1 if $\Delta = 0$, otherwise 0.

Fig. 3: The Inner Product Verification Protocol

Dimension reduction. We extend the dimension reduction technique of Goyal *et al.* [14] to our 3PC over ring setting. As shown in Fig. 2, protocol Π_{Reduce} takes a shared triple $(\{\langle x^{(i)} \rangle^{\ell[x]}, \langle y^{(i)} \rangle^{\ell[x]}, \langle z \rangle^{\ell[x]}\}_{i \in \mathbb{Z}_{|\mathcal{G}|}})$ as input and outputs $(\{\langle x'^{(i)} \rangle^{\ell[x]}, \langle y'^{(i)} \rangle^{\ell[x]}, \langle z' \rangle^{\ell[x]}\}_{i \in \mathbb{Z}_{|\mathcal{G}|/2}})$. Π_{Reduce} ensures that $\sum_{i=0}^{|\mathcal{G}|-1} x^{(i)} \cdot y^{(i)} = z$ if and only if $\sum_{i=0}^{|\mathcal{G}|/2-1} x'^{(i)} \cdot y'^{(i)} = z'$ except for a negligible probability. At a high level, for the inner product input $\{x^{(i)}\}_{i \in \mathbb{Z}_{|\mathcal{G}|}}$ and $\{y^{(i)}\}_{i \in \mathbb{Z}_{|\mathcal{G}|}}$, we can utilize $x^{(2i)}$ and $x^{(2i-1)}$ to interpolate $|\mathcal{G}|/2$ linear functions $\{f_i(\cdot)\}_{i \in \mathbb{Z}_{|\mathcal{G}|/2}}$ at the point 0 and 1, and similarly interpolate $\{g_i(\cdot)\}_{i \in \mathbb{Z}_{|\mathcal{G}|/2}}$ by $\{y^{(i)}\}_{i \in \mathbb{Z}_{|\mathcal{G}|}}$. Considering the correct output z , we have

$$z = \sum_{i=0}^{|\mathcal{G}|/2} f_i(0) \cdot g_i(0) + f_i(1) \cdot g_i(1)$$

Let $h(\cdot) = \sum_{i=0}^{|\mathcal{G}|/2} f_i(\cdot) \cdot g_i(\cdot)$. This leads to the equation $h(1) = z - h(0)$. The protocol Π_{Reduce} computes $h(0) = \sum_{i=0}^{|\mathcal{G}|/2} f_i(0) \cdot g_i(0)$ and $h(2) = \sum_{i=0}^{|\mathcal{G}|/2} f_i(2) \cdot g_i(2)$, and from this, it calculates $h(1) = z - h(0)$. Then, Π_{Reduce} interpolates the polynomial $h(x)$ using the values $h(0)$, $h(1)$, and $h(2)$. Finally, all parties choose a random point ζ and output the new shared triple $(\{\langle f_i(\zeta) \rangle^{\ell[x]}, \langle g_i(\zeta) \rangle^{\ell[x]}, \langle h(\zeta) \rangle^{\ell[x]}\}_{i \in \mathbb{Z}_{|\mathcal{G}|/2}})$, which preserves the inner product relation if and only if the initial condition $z = \sum_{i=1}^{|\mathcal{G}|/2} f_i(0) \cdot g_i(0) + f_i(1) \cdot g_i(1)$ holds.

It is important to note that the points 0, 1, and 2 correspond to ring elements with free coefficients of 0, 1, and 2 in $\mathbb{Z}_{2^\ell}[x]/f(x)$.

The protocol Π_{Reduce} requires one round of communication involving $5\ell \cdot d$ bits in the online phase and one round involving $\ell \cdot d$ bits in the offline phase. We execute Π_{Reduce} R times to reduce the inner product dimension to $|\mathcal{G}|/2^R$, after which the resulting vectors are verified by checking

$$\sum_{i=0}^{|\mathcal{G}|/2^R} \langle f_i(\zeta) \rangle^{\ell[x]} \cdot \langle g_i(\zeta) \rangle^{\ell[x]} = \langle h(\zeta) \rangle^{\ell[x]}$$

We prove the soundness error of the Π_{Reduce} is $\frac{1}{2^{d-1}}$ in Lemma 2.

Lemma 2. Suppose Π_{Reduce} take $(\{\langle x^{(i)} \rangle^{\ell[x]}, \langle y^{(i)} \rangle^{\ell[x]}, \langle z \rangle^{\ell[x]}\}_{i \in \mathbb{Z}_{|\mathcal{G}|}})$ as input, and it outputs the new list $(\{\langle x'^{(i)} \rangle^{\ell[x]}, \langle y'^{(i)} \rangle^{\ell[x]}, \langle z' \rangle^{\ell[x]}\}_{i \in \mathbb{Z}_{|\mathcal{G}|/2}})$. The

Protocol $\Pi_{\text{MultVerify}}^R(\{\langle x^{(i)} \rangle, \langle y^{(i)} \rangle, \langle z^{(i)} \rangle\}_{i \in \mathbb{Z}_{|\mathcal{G}|}})$

Input : $|\mathcal{G}|$ pairs of $\langle \cdot \rangle$ -shared multiplication.

Output : $z^{(i)} \stackrel{?}{=} x^{(i)} \cdot y^{(i)}$ for all $i \in \mathbb{Z}_{|\mathcal{G}|}$.

Execution:

- All parties invoke $\Pi_{\text{Trans}}(\{\langle x^{(i)} \rangle, \langle y^{(i)} \rangle, \langle z^{(i)} \rangle\}_{i \in \mathbb{Z}_{|\mathcal{G}|}})$ to get $\{\langle x^{(i)} \rangle^{\ell[x]}, \langle y^{(i)} \rangle^{\ell[x]}, \langle z \rangle^{\ell[x]}\}_{i \in \mathbb{Z}_{|\mathcal{G}|}}$;
- For $k = 1, \dots, R$, all parties perform:
 - $\{\{\langle x^{(i)} \rangle^{\ell[x]}, \langle y^{(i)} \rangle^{\ell[x]}, \langle z \rangle^{\ell[x]}\}_{i \in \mathbb{Z}_{|\mathcal{G}|/2^k}}\} \leftarrow \Pi_{\text{Reduce}}(\{\langle x^{(i)} \rangle^{\ell[x]}, \langle y^{(i)} \rangle^{\ell[x]}, \langle z \rangle^{\ell[x]}\}_{i \in \mathbb{Z}_{|\mathcal{G}|/2^{k-1}}})$;
- All parties invoke $b = \Pi_{\text{InnerVerify}}(\{\langle x^{(i)} \rangle^{\ell[x]}, \langle y^{(i)} \rangle^{\ell[x]}, \langle z \rangle^{\ell[x]}\}_{i \in \mathbb{Z}_{|\mathcal{G}|/2^R}})$;
- All parties output b .

Fig. 4: The Batch Multiplication Verification Protocol

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. $\mathbb{Z}_{2^\ell}[x]/f(x)$:

- $z' = \sum_{i=0}^{|\mathcal{G}|/2} x'^{(i)} \cdot y'^{(i)}$
- $z \neq \sum_{i=0}^{|\mathcal{G}|} x^{(i)} \cdot y^{(i)}$

Proof. For clarity, we denote $h'(k) = \sum_{i=0}^{|\mathcal{G}|/2} f_i(k) \cdot g_i(k)$. The adversary's goal is to manipulate the computation such that $h(\zeta) = h'(\zeta)$, while also ensuring that

$$h(0) + h(1) = h'(0) + h'(1) + e,$$

where e represents the error introduced in z . Simultaneously, the adversary can introduce new errors e_1 and e_2 during the calculation of $h(0)$ and $h(2)$, such that:

$$h(0) = h'(0) + e_1, \quad h(1) = h'(1) + e - e_1, \quad h(2) = h'(2) + e_2.$$

Considering the Lagrange interpolation for randomly chosen $\zeta \in \mathbb{Z}_{2^\ell}[x]$, we have:

$$h(\zeta) = \sum_{i=0}^2 \left(\prod_{\substack{j=0 \\ j \neq i}}^2 \frac{\zeta - j}{i - j} \right) \cdot h(i) = \frac{(\zeta - 1)(\zeta - 2)}{2} \cdot h(0) + \zeta(2 - \zeta) \cdot h(1) + \frac{(\zeta - 1)\zeta}{2} \cdot h(2)$$

and for $h'(\zeta)$:

$$h'(\zeta) = \frac{(\zeta - 1)(\zeta - 2)}{2} \cdot h'(0) + \zeta(2 - \zeta) \cdot h'(1) + \frac{(\zeta - 1)\zeta}{2} \cdot h'(2).$$

To ensure $h(\zeta) = h'(\zeta)$, the adversary must satisfy the following equation:

$$\frac{(\zeta - 1)(\zeta - 2)}{2} \cdot e_1 + \zeta(2 - \zeta) \cdot (e - e_1) + \frac{(\zeta - 1)\zeta}{2} \cdot e_2 = 0$$

The probability that the adversary can choose e, e_1, e_2 such that this equation holds is equivalent to making ζ a root of the degree-2 polynomial:

$$f(x) = \frac{(x - 1)(x - 2)}{2} \cdot e_1 + x(2 - x) \cdot (e - e_1) + \frac{(x - 1)x}{2} \cdot e_2$$

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

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 $\{\langle x^{(i)} \rangle, \langle y^{(i)} \rangle, \langle z^{(i)} \rangle\}_{i \in \mathbb{Z}_{|\mathcal{G}|}}$, all parties call $\Pi_{\text{MultVerify}}^R(\{\langle x^{(i)} \rangle, \langle y^{(i)} \rangle, \langle z^{(i)} \rangle\}_{i \in \mathbb{Z}_{|\mathcal{G}|}})$ to verify correctness.

Fig. 5: The Multiplication Protocol

over $\mathbb{Z}_{2^\ell}[x]$, which has at most $2^{2(\ell-1)d} + 1$ roots. Therefore, the soundness error is:

$$\frac{2^{(\ell-1)d+1} + 1}{2^{\ell d}} \approx \frac{1}{2^{d-1}}.$$

□

Inner product verification. Our inner product verification protocol, denoted as $\Pi_{\text{InnerVerify}}$ (Fig. 3), verifies the inner product relationship of shared values over the polynomial ring $\mathbb{Z}_{2^\ell}[x]/f(x)$. Specifically, to verify the relation

$$\sum_{i=0}^{|\mathcal{G}|/2^R} \langle x^{(i)} \rangle^{\ell[x]} \cdot \langle y^{(i)} \rangle^{\ell[x]} = \langle z \rangle^{\ell[x]},$$

$\Pi_{\text{InnerVerify}}$ checks whether the expression

$$\langle \alpha \rangle^{\ell[x]} \cdot \left(\sum_{i=0}^{|\mathcal{G}|/2^R} \langle x^{(i)} \rangle^{\ell[x]} \cdot \langle y^{(i)} \rangle^{\ell[x]} - \langle z \rangle^{\ell[x]} \right)$$

is equal to zero.

Unfortunately, as far as we know, there is currently no semi-honest 3PC protocol that securely evaluates a cubic (degree-3) polynomial while being resilient to additive attacks. As an alternative, we compute $x'^{(i)} = \alpha \cdot x^{(i)}$ for each $i \in \mathbb{Z}_{|\mathcal{G}|}$. Subsequently, all parties evaluate the inner product

$$\sum_{i=0}^{|\mathcal{G}|/2^R} x'^{(i)} \cdot y^{(i)}.$$

This method, however, does not achieve complete security against additive attacks, as an adversary may introduce an error $e'^{(i)}$ into $x'^{(i)}$, resulting in an overall error term:

$$\sum_{i=0}^{|\mathcal{G}|/2^R} e'^{(i)} \cdot y^{(i)},$$

which is dependent on $y^{(i)}$. Nevertheless, considering that $y^{(i)}$ is obtained via multiple Lagrange interpolations in the prior dimension reduction protocol, $y^{(i)}$ can be treated as a random value.

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

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

Postprocessing:

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

Fig. 6: The Inner Product Protocol

Let e denote the error in z . The adversary must guess $\alpha \cdot e + \sum_{i=0}^{|\mathcal{G}|/2^R} e'^{(i)} \cdot y^{(i)} = 0$, where $y^{(i)}$ is effectively random. The probability of success for this guess is $\frac{1}{2^d}$.

We prove in Lemma 3 that the soundness error of the $\Pi_{\text{InnerVerify}}$ protocol is $\frac{1}{2^d}$.

Lemma 3. *Let $(\{\langle x^{(i)} \rangle^{\ell[x]}, \langle y^{(i)} \rangle^{\ell[x]}\}_{i \in \mathbb{Z}_{|\mathcal{G}|}}, \langle z \rangle^{\ell[x]})$ be the input of protocol $\Pi_{\text{InnerVerify}}$ depicted in Fig. 3. The probability that $\Pi_{\text{InnerVerify}}$ outputs 1 and $z \neq \sum_{i=0}^{|\mathcal{G}|/2^R} x^{(i)} \cdot y^{(i)}$ 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 $z = \sum_{i=0}^{|\mathcal{G}|/2^R} x^{(i)} \cdot y^{(i)} + e$, we have

$$\Delta = \alpha \cdot e + \sum_{i=0}^{|\mathcal{G}|/2^R} e'^{(i)} \cdot y^{(i)},$$

where $e'^{(i)}$ is introduced during the evaluation of $\alpha \cdot x^{(i)}$. Given that 3PC multiplication is secure up to additive attacks, $e'^{(i)}$ is independent of α . Therefore, we can treat $\sum_{i=0}^{|\mathcal{G}|/2^R} e'^{(i)} \cdot y^{(i)}$ as an overall error term e' .

By the Schwartz-Zippel Lemma, the polynomial $f(x) = e \cdot x + e'$ over the ring $\mathbb{Z}_{2^\ell}[x]$ has at most $2^{(\ell-1)d} + 1$ roots. Consequently, the probability that the adversary can deliberately choose e such that $\Delta = 0$ is

$$\frac{2^{(\ell-1)d} + 1}{2^{\ell d}} \approx \frac{1}{2^d}.$$

□

Our batch multiplication verification protocol $\Pi_{\text{MultVerify}}$ in Fig. 4 integrates the above three subroutines, which requires one round communication of $(R+|\mathcal{G}|/2^R)\ell \cdot d$ bits in the offline phase and $R+2$ -round communication of $(5R+3+|\mathcal{G}|/2^R)\ell \cdot d$ bits in the online phase for $|\mathcal{G}|$ multiplication triples. We prove soundness error of $\Pi_{\text{MultVerify}}$ is $\frac{|\mathcal{G}|}{2^{d-R-2}}$ in Thm. 1.

Theorem 1. *Let $\{\langle x^{(i)} \rangle, \langle y^{(i)} \rangle, \langle z^{(i)} \rangle\}_{i \in \mathbb{Z}_{|\mathcal{G}|}}$ be the input of protocol $\Pi_{\text{MultVerify}}^R$ depicted in Fig. 4. The probability*

Protocol $\Pi_{\text{BIVerify}}^R(\{\langle x_i^{(j)} \rangle, \langle y_i^{(j)} \rangle\}_{i \in \mathbb{Z}_{n_j}}, \langle z^{(j)} \rangle\}_{j \in \mathbb{Z}_{|\mathcal{G}|}})$

Input : $|\mathcal{G}|$ pairs of inner product.

Output : Output if $z^{(j)} = \sum_{i=1}^n x_i^{(j)} \cdot y_i^{(j)}$ held for all $j \in \mathbb{Z}_{|\mathcal{G}|}$.

Execution:

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

Fig. 7: The Batch Inner Product Verification Protocol

$\Pi_{\text{MultVerify}}^R$ outputs 1 and $\exists i \in \mathbb{Z}_{|\mathcal{G}|}$ s.t. $z^{(i)} \neq x^{(i)} \cdot y^{(i)}$ is at most $\frac{|\mathcal{G}|}{2^{d-R-2}}$, 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 success probability $\frac{1}{2^{d-1}}$, one chance with probability $\frac{|\mathcal{G}|}{2^d}$, and one chance with probability $\frac{1}{2^d}$ to pass the verification.

Therefore, the total probability that the adversary succeeds is

$$1 - \left(1 - \frac{1}{2^{d-1}}\right)^R \cdot \left(1 - \frac{|\mathcal{G}|}{2^d}\right) \cdot \left(1 - \frac{1}{2^d}\right) \approx \frac{|\mathcal{G}|}{2^{d-R-2}}.$$

□

IV. ENHANCING PPML.

In this section, we implement a maliciously secure privacy-preserving machine learning framework. We use boolean share to evaluate nonlinear functions, which can be viewed as share over ring \mathbb{Z}_2 . We realize the share conversion protocol, which is entirely based on maliciously secure multiplication Π_{Mult} . This makes our framework merely reliant on Π_{Mult} .

A. Dealing with linear operation.

Our maliciously secure multiplication protocol is shown in Fig. 5. Π_{Mult} ensures the correctness of multiplication by invoking batch verification protocol $\Pi_{\text{MultVerify}}$ 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. Our maliciously secure inner product protocol Π_{Inner} is shown in Fig. 6. 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

Protocol $\Pi_{\text{Trunc}}^t(\langle x \rangle)$

Let $\text{rshift}(x, y)$ denote right shift x with y bits. Input : $\langle \cdot \rangle$ -shared value.

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

Preprocessing:

- P_0 and P_i pick random bit list $\{b_{i,j}\}_{j \in \mathbb{Z}_\ell} \leftarrow \mathbb{Z}_2^\ell$ 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 $m_z = \text{rshift}(m_x, t);$
- All parties output $\langle z \rangle := ([r_z], m_z).$

Fig. 8: The maliciously secure truncation protocol

round communication of 2ℓ bits in the online phase. To extend it to the malicious setting, we employ batch verification protocol $\Pi_{\text{InnerVerify}}^R$ (Fig. 7) to ensure the correctness of the inner products with a similar manner of multiplication. Analogously, in $\Pi_{\text{InnerVerify}}^R$, all parties transform the verification of inner product triples over ring \mathbb{Z}_{2^ℓ} to the verification of a single inner product triple over the polynomial ring $\mathbb{Z}_{2^\ell}[x]/f(x)$. Following that, all parties invoke Π_{Reduce} to reduce the dimension of the vector that needs to be verified. When handling a substantial volume of data, on average, our protocol exhibits an amortized communication of ℓ bits in the preprocessing phase and 2ℓ bits in the online phase for each inner product operation. In the application of machine learning, we view the m -dimensional output convolution and matrix multiplication as m separate inner products. We implement these two types of operations by invoking Π_{Inner} a total of m times.

B. Secure Truncation Protocol.

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 [4], [5], [12] 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. 8, we also design a maliciously secure probabilistic truncation protocol Π_{Trunc}^t for the truncation bit size t . Our idea is similar to SWIFT [12], which generates correct truncation pair via maliciously secure inner product protocol. However, in contrast to SWIFT [12], we directly generate $r_z = \text{rshift}(r_x, d)$, which allows the

Protocol Π_{edaBits}

Input : None.

Output : edaBits pair $\{\langle r \rangle^\ell, \{\langle r[i] \rangle^1\}_{i \in [\ell]}\}$.

Execution:

- P_0 and P_j pick random bit list $\{b_{i,j}\}_{i \in [\ell]} \leftarrow (\mathbb{Z}_2)^\ell$ together, for $i \in \{1, 2\}$;
- P_1 and P_2 pick random bit list $\{m_i\}_{i \in [\ell]} \leftarrow (\mathbb{Z}_2)^\ell$ together, for $i \in \{1, 2\}$;
- All parties set $\langle r[i] \rangle^1 := (m_i, b_{i,1}, b_{i,2})$;
- All parties set
 - $\langle b_{i,1} \rangle^\ell := (0, b_{i,1}, 0)$;
 - $\langle b_{i,2} \rangle^\ell := (0, 0, b_{i,2})$;
 - $\langle m_i \rangle^\ell := (m_i, 0, 0)$ for $i \in [\ell]$;
- All parties invoke Π_{Mult} to calculate
 - $\langle r'[i] \rangle^\ell = \langle b_{i,1} \rangle^\ell + \langle b_{i,2} \rangle^\ell - 2\langle b_{i,1} \rangle^\ell \cdot \langle b_{i,2} \rangle^\ell$ for $i \in [\ell]$;
 - $\langle r \rangle^\ell = \sum_{i=0}^{\ell-1} 2^i \cdot (\langle m_i \rangle^\ell + \langle r'[i] \rangle^\ell - 2\langle m_i \rangle^\ell \cdot \langle r'[i] \rangle^\ell)$
- All parties output $\{\langle r \rangle^\ell, \{\langle r[i] \rangle^1\}_{i \in [\ell]}\}$

Fig. 9: The maliciously edaBits generation

parties locally truncate $m_z = \text{rshift}(m_x, d)$ in the online phase without communication. Although SWIFT [12] 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 $\{b_{1,j}\}_{j \in \mathbb{Z}_\ell}$ together; P_0 and P_2 pick random bit list $\{b_{2,j}\}_{j \in \mathbb{Z}_\ell}$ 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 $r_z = \sum_{j=0}^{\ell-1} 2^j \cdot (b_{1,j} \oplus b_{2,j}) + \sum_{j=\ell-t-1}^{\ell-1} 2^j \cdot (b_{1,\ell-1} \oplus b_{2,\ell-1})$ which keeps the relationship $r_z = \text{shift}(r_x, t)$. 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.

C. Secure Non-linear Function Evaluation.

We utilize standard daBits [26], [31] to evaluate non-linear functions. DaBits convert arithmetic secret sharing into Boolean secret sharing, allowing us to use Boolean shares for circuit evaluation corresponding to the non-linear function. In our protocol, a daBit consists of a pair $\langle r \rangle^\ell, \langle r \rangle^1$, where $r \in \{0, 1\}$. As a multi-bit version, edaBits [31] is a pair $\{\langle r \rangle^\ell, \langle r[i] \rangle^1\}_{i \in [\ell]}$, where $r \in \mathbb{Z}_2$ is an arithmetic share, and $\langle r[i] \rangle^1$ are Boolean shares resulting from bit extraction. We demonstrate that our maliciously secure 3PC protocol is fully compatible with edaBits.

EdaBits allow bit extraction from shares over \mathbb{Z}_{2^ℓ} into multiple shares over \mathbb{Z}_2 . A series of works [4] use full adders to perform this conversion. Specifically, let $\langle x \rangle^\ell$ be the share requiring bit extraction and $\{\langle r \rangle^\ell, \{\langle r[i] \rangle^1\}_{i \in [\ell]}\}$

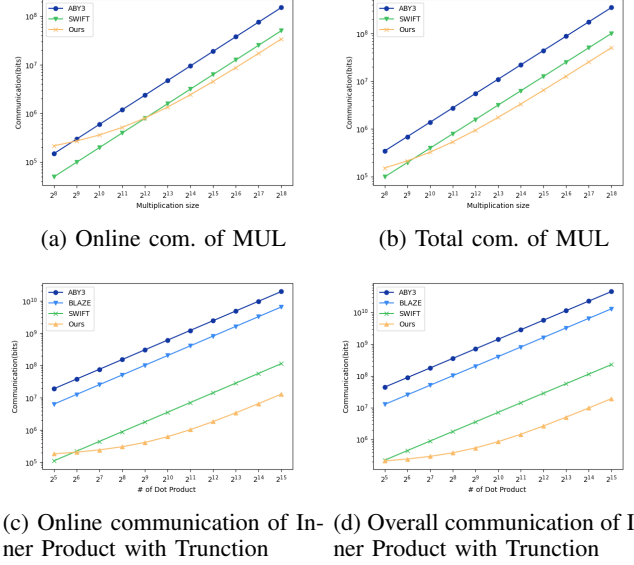


Fig. 10: Communication overhead comparison with ABY3 [4], BLAZE [5], SWIFT [12] of multiplication and inner product.

be the edaBits. All parties first reconstruct $\Delta = x - r$ by locally computing $\langle \Delta \rangle^\ell = \langle x \rangle^\ell - \langle r \rangle^\ell$. Subsequently, using $\{\langle r[i] \rangle^1\}_{i \in [\ell]}$ and $\{\langle \Delta[i] \rangle^1\}_{i \in [\ell]}$, the parties perform a full-adder circuit to obtain $\{\langle x[i] \rangle^1\}_{i \in [\ell]}$. Using $x[i]$, each party can then evaluate arbitrary non-linear functions. Note that the soundness of our malicious 3PC protocol is independent of the ring size ℓ , and it works for $\ell = 1$ as well.

For the share conversion from $\langle x \rangle^1$ to $\langle x \rangle^\ell$, this can be achieved through multiplication. Given the edaBits $\{\langle r \rangle^\ell, \langle r \rangle^1\}$, all parties first reconstruct $\Delta = \langle x \rangle^1 - \langle r \rangle^1$ over \mathbb{Z}_2 . Then, they compute $\langle x \rangle^\ell = \Delta + \langle r \rangle^\ell - 2\Delta \cdot \langle r \rangle^\ell$. This works because $\Delta \oplus r = \Delta + r - 2\Delta \cdot r$.

The edaBits $\{\langle r \rangle^\ell, \{\langle r[i] \rangle^1\}_{i \in [\ell]}\}$ can be constructed from multiple daBits $\{\langle r[i] \rangle^\ell, \{\langle r[i] \rangle^1\}_{i \in [\ell]}\}$ by computing $\langle r \rangle^\ell = \sum_{i=0}^{\ell-1} 2^i \cdot \langle r[i] \rangle^\ell$.

To generate daBits $\{\langle r \rangle^\ell, \langle r \rangle^1\}$, parties P_0 and P_1 jointly pick a random value $r_1 \in \{0, 1\}$, P_0 and P_2 pick $r_2 \in \{0, 1\}$, and P_1 and P_2 pick $r_3 \in \{0, 1\}$. All parties then set $\langle r_1 \rangle^\ell := (0, r_1, 0)$, $\langle r_2 \rangle^\ell := (0, 0, r_2)$, and $\langle r_3 \rangle^\ell := (r_3, 0, 0)$. The parties set $\langle r \rangle^1 := (r_3, r_1, r_2)$, and compute $\langle r \rangle^\ell = \langle r_1 \rangle^\ell + \langle r_2 \rangle^\ell - 2\langle r_1 \rangle^\ell \cdot \langle r_2 \rangle^\ell$. Finally, they compute $\langle r \rangle^\ell = \langle r \rangle^\ell + \langle r_3 \rangle^\ell - 2\langle r \rangle^\ell \cdot \langle r_3 \rangle^\ell$.

Note that $\langle r \rangle^\ell$ requires two rounds of multiplication. Considering edaBits $\{\langle r \rangle^\ell, \{\langle r[i] \rangle^1\}_{i \in [\ell]}\}$, the second multiplication can be combined with a single inner product. Fig. 9 depicts the generation of edaBits.

Our PPML framework is constructed by multiplication over \mathbb{Z}_{2^ℓ} and \mathbb{Z}_2 , which can be verified by perform $\Pi_{\text{MultVerify}}$ on \mathbb{Z}_{2^ℓ} and \mathbb{Z}_2 respectively.

V. IMPLEMENTATION AND BENCHMARKS

In this section, we evaluate our multiplication and non-linear protocols in both the semi-honest and malicious settings. For the maliciously secure multiplication protocols, we compare the communication and runtime with SWIFT [12] and ABY [4].

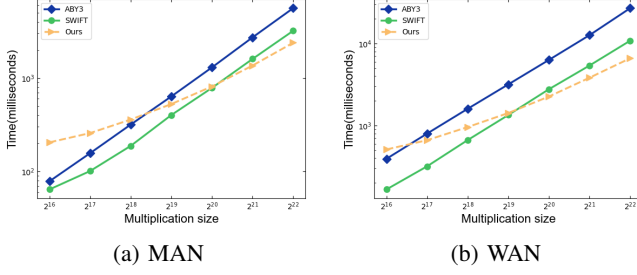


Fig. 11: Overall running time of multiplication (over the GPU setting). Compared with ABY3 [4], SWIFT [12] of Π_{Mult} over MAN and WAN setting.

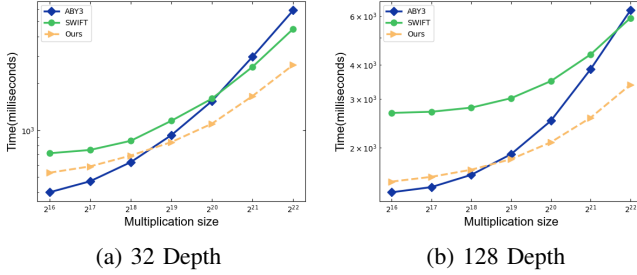


Fig. 12: Evaluate the multiplication (over the GPU setting) with circuit depth 32 and 128 under the MAN setting.

Benchmark setting. We perform our arithmetic protocols on the GPU setting. To support GPU, our code is based on the Piranha [27] source code [32], which is a GPU platform for MPC protocols. For the non-linear protocols, we implement both CPU and GPU versions to support benchmarking with FSS [33] and garble circuit-based protocol BLAZE [5] on CPU setting. The modified version of Piranha (GPU-version) [34], and the CPU version of our code [35] are available in Anonymize Github. 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.

A. Multiplication performance comparison

In this section, we benchmark our maliciously secure 3PC of multiplication.

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. 13 depicts the verification

time with the different dimension reduction number R . The running time is measured in milliseconds on the y-axis, while the x-axis shows the number of dimension reductions R , ranging from 0 to 10. 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.

Communication. Fig. 10 presents the communication overhead of our protocol compared to ABY, BLAZE, and SWIFT. For this evaluation, we consider a vector dimension of 1024 when computing the inner product. Our protocol incurs a logarithmic additional communication cost of $(6R + 5)\ell \cdot d$, where $R = \log |\mathcal{G}|$. As a result, when $|\mathcal{G}|$ is small, our protocol requires more communication than SWIFT. However, as $|\mathcal{G}|$ increases, the logarithmic term R diminishes in relative significance, making the additional overhead negligible for large $|\mathcal{G}|$. In scenarios with large input sizes, the communication volume of our protocol for multiplication is approximately 50% that of SWIFT and 15% that of ABY. For the inner product computation in 1024 dimensions with truncation, the communication overhead is about 50% that of SWIFT and 0.02% that of ABY. This demonstrates that while our protocol may initially have higher communication costs for smaller input sizes, it becomes significantly more efficient as the input size increases. Specifically, it provides notable savings in communication compared to both SWIFT and ABY in large-scale computations, particularly in higher-dimensional inner product operations.

Performance. We compare our maliciously secure multiplication protocol with state-of-the-art (SOTA) protocols in Fig. 11, including SWIFT and ABY. To assess the running time, we execute our protocol across multiple values of R (the dimension reduction factor), selecting the configuration that yields the best performance. Due to the inclusion of an additional verification round—especially in scenarios with a small volume of data—our protocol experiences a performance overhead, making it slower than SWIFT and ABY in these cases. This verification round is the dominant source of overhead when the data size is small. However, when considering larger, saturated datasets, our protocol demonstrates a significant performance improvement, achieving up to $2\times$ the throughput of both SWIFT and ABY under both MAN (Metropolitan Area Network) and WAN (Wide Area Network) settings.

Furthermore, we investigate the effect of multiplication depth on protocol performance. Fig. 12 illustrates how performance varies with different multiplication depths. We benchmark our protocol against others on multiplication circuits with depths of 32 and 128. Due to the ability of our protocol and ABY to leverage batch verification, they exhibit a round complexity advantage, resulting in better performance compared to the SWIFT protocol, especially when the multiplication depth is large. This round reduction becomes particularly beneficial as circuit depth increases, allowing our protocol to outperform SWIFT in deep circuit scenarios.

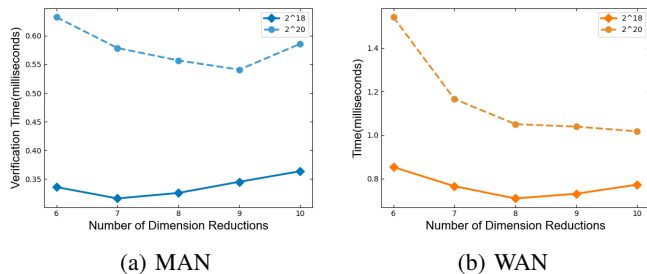


Fig. 13: The running time of verification phase (over the GPU setting), with the different dimension reduction number R , multiplication triple size 2^{18} and 2^{20} , over MAN and WAN setting.

TABLE II: 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	3	48.2
VGG	Execution	10.2	207	39.2	127	1341
	Verification	-	-	414	6	257

In summary, while our protocol may incur some overhead in low-volume, low-depth scenarios due to the additional verification step, it scales efficiently with larger datasets and deeper circuits, making it highly competitive against SOTA protocols such as SWIFT and ABY in high-volume, high-depth applications.

B. 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 II presents the performance metrics of our protocol across different stages for three models: S-NN, LeNet, and VGG. The metrics are divided into communication (Com) and time (Time). During the execution stage, all parties engage in both the offline and online phases of the semi-honest protocol, ensuring the necessary computations are securely executed. The communication costs rise significantly with model complexity, especially in the execution phase. In the verification stage, a post-processing step is performed by all parties to validate the correctness of the computed and shared results. Our platform demonstrates the capability to execute CNNs, such as LeNet, in mere hundreds of milliseconds. For deeper convolutional networks, like VGG, the platform completes the execution within seconds, showcasing its efficiency even with more complex neural network models. As observed in the provided table, when the model is small, such as in the case of the S-NN model, the verification phase takes a significant proportion of the total runtime compared to the online execution phase. However, as the model complexity increases, such as with LeNet and VGG, the runtime required for the online execution phase grows substantially while the proportion of the verification phase decreases. In summary, for small models, verification is a major contributor to total runtime. However, as the model size increases, the overhead of verification decreases in proportion and becomes less impactful, with the online phase becoming the dominant factor in overall runtime.

Fig. 14 illustrates the computation and communication latencies during the model inference process. The figure details the latencies for two models, LeNet and VGG, across the execution phase (Exec) and the verification phase (Verify). The results highlight that, during the execution phase, communication latency far exceeds computation latency. This effect is particularly pronounced for LeNet, where communication latency dominates the entire inference process, underscoring the significant impact of network communication on overall performance. In contrast, during the verification phase, computation latency becomes the primary factor contributing to the total latency. As the complexity of the model increases, as seen when comparing VGG to LeNet, the communication overhead becomes the main bottleneck in the online phase, severely impacting inference efficiency. Meanwhile, in the verification phase, the computation overhead takes precedence, becoming the dominant factor that limits overall performance. These findings emphasize the need for optimizing both communication and computation aspects, particularly as model complexity increases, to improve inference efficiency across all stages.

VI. RELATED WORK

[13] 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. [4]–[6], [9], [11], [11], [12], [24], [25] realize privacy-preserving machine learning protocols under the malicious threat model in an honest majority. In the semi-honest setting, protocols such as [4], [5], [11], [36] are all

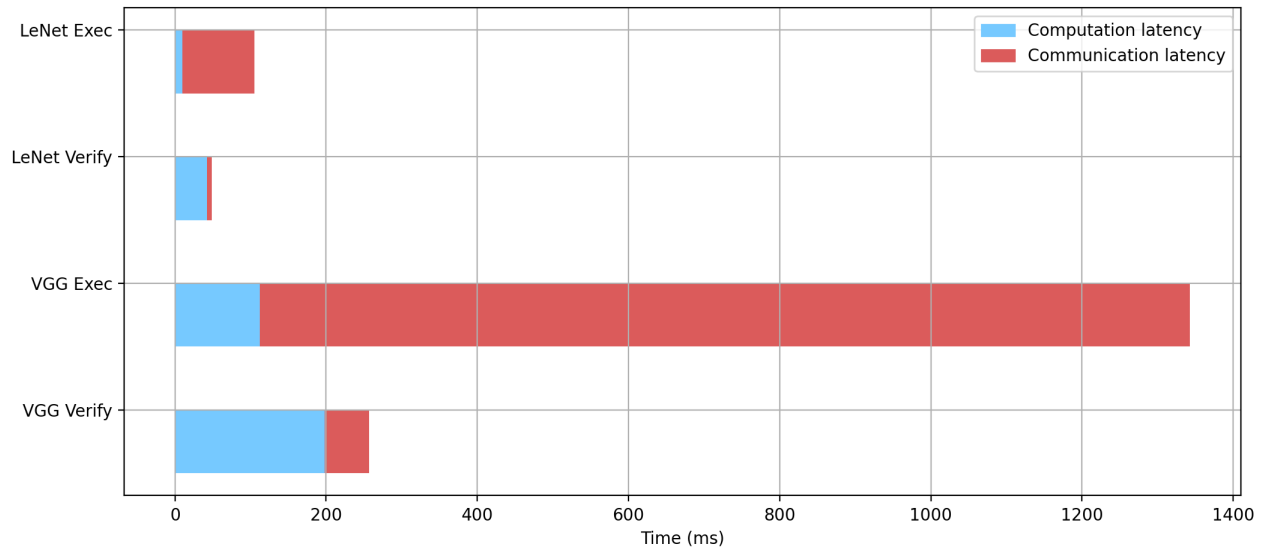


Fig. 14: Computation latency and communication latency diagram for model inference in LAN setting. Exec corresponds to the execution phase (both offline and online), and Verify corresponds to the verification phase.

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 [11]. For malicious security, 3PC and 4PC protocols with an honest majority, such as ASTRA [11], SWIFT [12], and others [4], [5], [37], [38] have been developed for private machine learning and advanced applications. These protocols combine function-dependent pre-processing and mixed-protocol strategies, providing malicious security while optimizing performance. In particular, a series of optimizations [5], [11], [12] reduced the multiplication overhead to 6 ring elements (3 in the offline phase) in the three-party setting. Other notable contributions include efforts to reduce the cost of garbled circuits against a malicious evaluator [39] and improving communication efficiency in settings with a dishonest majority and semi-honest helper party [40]. Additionally, protocols like MUSE [41] and follow-up works [42], [43] explored fixed-corruption scenarios in 2PC, enhancing malicious security while reducing performance overhead.

VII. CONCLUSION

In this work, we design a 3PC maliciously secure multiplication protocol over the ring. We adopt our protocol for the machine learning model evaluation and perform the model evaluation on the GPU platform. The experiments show that our various protocols have significant performance improvements over the state-of-the-art works, i.e., [4], [12].

REFERENCES

- [1] A. C. Yao, "Protocols for secure computations," in *SFCS*, 1982.
- [2] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game," in *STOC*, 1987.
- [3] D. Beaver, "Efficient multiparty protocols using circuit randomization," in *CRYPTO*, 1991.
- [4] P. Mohassel and P. Rindal, "Aby3: A mixed protocol framework for machine learning," in *CCS*, 2018.
- [5] A. Patra and A. Suresh, "BLAZE: blazing fast privacy-preserving machine learning," in *NDSS*, 2020.
- [6] M. Byali, H. Chaudhari, A. Patra, and A. Suresh, "Flash: Fast and robust framework for privacy-preserving machine learning," in *PoPETs*, 2020.
- [7] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, "Chameleon: A hybrid secure computation framework for machine learning applications," in *ASIACCS*, 2018.
- [8] E. Makri, D. Rotaru, N. P. Smart, and F. Vercauteren, "Epic: Efficient private image classification (or: Learning from the masters)," in *CT-RSA*, 2019.
- [9] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin, "Falcon: Honest-majority maliciously secure framework for private deep learning," in *PoPETs*, 2021.
- [10] S. Wagh, D. Gupta, and N. Chandran, "Securenn: 3-party secure computation for neural network training," in *PoPETs*, 2019.
- [11] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh, "Astra: High throughput 3pc over rings with application to secure prediction," in *CCSW*, 2019.
- [12] N. Koti, M. Pancholi, A. Patra, and A. Suresh, "Swift: Super-fast and robust privacy-preserving machine learning," in *USENIX*, 2021.
- [13] A. Dalskov, D. Escudero, and A. Nof, "Fast fully secure multi-party computation over any ring with two-thirds honest majority," in *CCS*, 2022.
- [14] V. Goyal and Y. Song, "Malicious security comes free in honest-majority mpc," Cryptology ePrint Archive, Paper 2020/134, 2020.
- [15] D. Escudero and V. Goyal, "Turbopack: Honest majority mpc with constant online communication," in *CCS*, 2022.
- [16] P. S. Nordholt and M. Veeningen, "Minimising communication in honest-majority mpc by batchwise multiplication verification," in *ACNS*, 2018.
- [17] I. Damgård, C. Orlandi, and M. Simkin, "Yet another compiler for active security or: Efficient mpc over arbitrary rings," in *CRYPTO*, 2018.
- [18] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing, "Spdz2k: Efficient MPC mod 2k for dishonest majority," in *CRYPTO*, 2018.
- [19] E. Hendrik, K. Marcel, O. Claudio, P. Pille, P. Joonas, and S. Mark, "Use your brain! arithmetic 3pc for any modulus with active security," in *ITC*, 2020.
- [20] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai, "Zero-knowledge proofs on secret-shared data via fully linear pcps," in *CRYPTO*, 2019.

- [21] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof, “Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs,” in *CCS*, 2019.
- [22] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof, “Sublinear gmw-style compiler for mpc with preprocessing,” in *Advances in Cryptology – CRYPTO 2021*, 2021.
- [23] W. Henecka, S. K ögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, “Tasty: Tool for automating secure two-party computations,” in *CCS*, 2010.
- [24] M. Payman and Z. Yupeng, “Secureml: A system for scalable privacy-preserving machine learning,” in *S&P*, 2017.
- [25] H. Chaudhari, R. Rachuri, and A. Suresh, “Trident: Efficient 4pc framework for privacy preserving machine learning,” in *NDSS*, 2020.
- [26] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl, “Improved primitives for mpc over mixed arithmetic-binary circuits,” in *Advances in Cryptology – CRYPTO 2020*, D. Micciancio and T. Ristenpart, Eds. Cham: Springer International Publishing, 2020, pp. 823–852.
- [27] J.-L. Watson, S. Wagh, and R. A. Popa, “Piranha: A gpu platform for secure computation,” 2022.
- [28] L. Yehuda and N. Ariel, “A framework for constructing fast mpc over arithmetic circuits with malicious adversaries and an honest-majority,” in *CCS*, 2017.
- [29] R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias, “Semi-homomorphic encryption and multiparty computation,” in *EUROCRYPT*, 2011.
- [30] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof, “Fast large-scale honest-majority mpc for malicious adversaries,” in *CRYPTO*, 2018.
- [31] D. Rotaru and T. Wood, “Marbled circuits: Mixing arithmetic and boolean circuits with active security,” in *INDOCRYPT*, 2019.
- [32] J.-L. Watson, S. Wagh, and R. A. Popa, “Piranha source code,” 2022. [Online]. Available: <https://github.com/ucbrise/piranha.git>
- [33] E. Boyle, N. Chandran, N. Gilboa, D. Gupta, Y. Ishai, N. Kumar, and M. Rathee, “Function secret sharing for mixed-mode and fixed-point secure computation,” in *EUROCRYPT*, 2021.
- [34] “Our modified piranha code.” [Online]. Available: <https://anonymous.4open.science/r/piranha-7B71>
- [35] “Our code.” [Online]. Available: https://anonymous.4open.science/r/oram_pro-CCD6
- [36] A. Patra, T. Schneider, A. Suresh, and H. Yalame, “Aby2. 0: Improved mixed-protocol secure two-party computation,” in *USENIX*, 2021.
- [37] A. Dalskov, D. E. Escudero, and M. Keller, “Fantastic four: Honest-majority four-party secure computation with malicious security,” *IACR Cryptol. ePrint Arch.*, 2020.
- [38] N. Koti, A. Patra, R. Rachuri, and A. Suresh, “Muse: Secure inference resilient to malicious clients,” in *USENIX Security*, 2021.
- [39] D. Evans, V. Kolesnikov, M. Rosulek *et al.*, “A pragmatic introduction to secure multi-party computation,” *Foundations and Trends® in Privacy and Security*, vol. 2, no. 2-3, pp. 70–246, 2018.
- [40] B. Karmakar, N. Koti, A. Patra, S. Patranabis, P. Paul, and D. Ravi, “Asterisk: Super-fast MPC with a friend,” *Cryptology ePrint Archive*, Paper 2023/1098, 2024. [Online]. Available: *S&P*
- [41] R. Lehmkuhl, P. Mishra, A. Srinivasan, and R. A. Popa, “Muse: Secure inference resilient to malicious clients,” in *USENIX Security*, 2021.
- [42] N. Chandran, D. Gupta, S. L. B. Obbattu, and A. Shah, “SIMC: ML inference secure against malicious clients at Semi-Honest cost,” in *USENIX Security*, 2022.
- [43] C. Dong, J. Weng, J.-N. Liu, Y. Zhang, Y. Tong, A. Yang, Y. Cheng, and S. Hu, “Fusion: Efficient and secure inference resilient to malicious servers,” in *Proceedings 2023 Network and Distributed System Security Symposium*, 2023.