Tetrad: Actively Secure 4PC for Secure Training and Inference

Nishat Koti*, Arpita Patra*, Rahul Rachuri†, Ajith Suresh*§
*Indian Institute of Science, Bangalore, Email: {kotis, arpita, ajith}@iisc.ac.in
†Aarhus University, Denmark, Email: rachuri@cs.au.dk

***Abstract*—Mixing arithmetic and boolean circuits to perform privacy-preserving machine learning has become increasingly popular. Towards this, we propose a framework for the case of four parties with at most one active corruption called Tetrad.**

Tetrad works over rings and supports two levels of security, fairness and robustness. The fair multiplication protocol costs 5 ring elements, improving over the state-of-the-art Trident (Chaudhari et al. NDSS'20). A key feature of Tetrad is that robustness comes for free over fair protocols. Other highlights across the two variants include (a) probabilistic truncation without overhead, (b) multi-input multiplication protocols, and (c) conversion protocols to switch between the computational domains, along with a tailor-made garbled circuit approach.

Benchmarking of Tetrad for both training and inference is conducted over deep neural networks such as LeNet and VGG16. We found that Tetrad is up to 4 times faster in ML training and up to 5 times faster in ML inference. Tetrad is also lightweight in terms of deployment cost, costing up to 6 times less than Trident.

I. Introduction

Increased concerns about privacy coupled with policies such as European Union General Data Protection Regulation (GDPR) make it harder for multiple parties to collaborate on machine learning computations. The emerging field of privacy-preserving machine learning (PPML) addresses this issue by offering tools to let parties perform computations without sacrificing privacy of the underlying data. PPML can be deployed across various domains such as healthcare, recommendation systems, text translation, etc., with works like [1] demonstrating practicality.

One of the main ways in which PPML is realised is through the paradigm of secure outsourced computation (SOC). Clients can outsource the training/prediction computation to powerful servers available on a 'pay-per-use' basis from cloud service providers. Of late, secure multiparty computation (MPC) based techniques [2]–[10] have been gaining interest, where a server enacts the role of a party in the MPC protocol. MPC [11], [12] allows mutually distrusting parties to compute a function in a distributed fashion while guaranteeing *privacy* of the parties' inputs and *correctness* of their outputs against any coalition of t parties.

The goal of PPML is practical deployment, making *efficiency* a primary consideration. Functions such as comparison, activation functions (e.g., ReLU), are heavily used in machine learning. Instantiating these functions via MPC naively turns out to be prohibitively inefficient due to their non-linearity. Hence, there is motivation to design specialised protocols that can compute these functions efficiently. We work towards this goal in the 4-party (4PC) setting, assuming honest majority [2], [4], [13], [14]. 4PC is interesting because it buys us the following over 3PC (which is threshold optimal): (1) *independence from broadcast:* broadcast can be achieved by a simple protocol in which the sender sends to everyone and residual parties exchange and apply a majority rule (2) *efficient dot-product:* 4PC offers a more efficient dot-product protocol (which is an important building block for several ML algorithms) with communication complexity independent of feature size (3) *simplicity and efficiency:* protocols are vastly more efficient and simpler in terms of design (as shown in this and prior works). To enhance practical efficiency, many recent works [4], [8], [15], [16] resort to the preprocessing paradigm, which splits the computation into two phases; a preprocessing phase where input-independent (but function-dependent) computationally heavy tasks can be computed, followed by a fast online phase. Since the same functions in ML are evaluated several times, this paradigm naturally fits the case of PPML, where the ML algorithm is known beforehand. Further, recent works [15], [17], [18] propose MPC protocols over 32 or 64 bit rings to leverage CPU optimizations.

MPC protocols can be categorized as high-throughput [3], [4], [6], [8], [14], [19]–[23] and low-latency [24], [25], where the former, based on secret-sharing, requires less communication compared to the latter (garbled circuits). High-throughput protocols typically work over the boolean ring $\mathbb{Z}_2$ or an arithmetic ring $\mathbb{Z}_{2^\ell}$ and aim to minimize communication overhead (bandwidth) at the expense of non-constant rounds. While high-throughput protocols enable efficient computation of functions such as addition, multiplication and dot-product, other functions such as division are best performed using garbled circuits. Activation functions such as ReLU used in neural networks (NN) alternate between multiplication and comparison, wherein multiplication is better suited to the arithmetic world and comparison to the boolean world. Hence, MPC protocols working over different representations (arithmetic/boolean/garbled circuit based) can be mixed to achieve better efficiency. This provided motivation for mixed protocols where each subprotocol is executed in a world where it performs best. Mixed-protocol frameworks [4], [6], [7], [10], [17], [23], [26], [27] have support for efficient ways to switch between the worlds, thereby getting the best from each of them. This work proposes a mixed-protocol PPML framework via MPC with four parties and honest majority with active security.

This article is the full and extended version of an article to appear in Network and Distributed System Security Symposium (NDSS) 2022.

§Corresponding Author. Currently postdoctoral researcher at TU Darmstadt (suresh@encrypto.cs.tu-darmstadt.de)

# Parties	Reference[a]	#Active Parties[b]	Security	Dot Product[c]		Dot Product with Truncation		Conversions[d]		
				Comm_{pre}[e]	Comm_{on}	Comm_{pre}	Comm_{on}	A	B	G
3	ABY3 [6]	3	Abort	$12\mathsf{d}\ell$	$9\mathsf{d}\ell$	$12\mathsf{d}\ell + 84\ell$	$9\mathsf{d}\ell + 3\ell$	✓	✓	✓
	BLAZE [8]	2	Fair	3ℓ	3ℓ	15ℓ	3ℓ	✓	✓	✗
	SWIFT (3PC) [14]	2	Robust	3ℓ	3ℓ	15ℓ	3ℓ	✓	✓	✗
4	Mazloom et al. [28]	4	Abort	2ℓ	4ℓ	2ℓ	4ℓ	✓	✓	✗
	Trident [4]	3	Fair	3ℓ	3ℓ	6ℓ	3ℓ	✓	✓	✓
	Tetrad	2	Fair	2ℓ	3ℓ	2ℓ	3ℓ	✓	✓	✓
	SWIFT (4PC) [14]	2	Robust	3ℓ	3ℓ	4ℓ	3ℓ	✓	✓	✗
	Fantastic Four [29] (Best)[f]	4	Robust	-	6ℓ	ℓ	9ℓ	✓	✓	✗
	Fantastic Four [29] (Worst)	3	Robust	-	$6(\ell + \kappa)$	$\approx 80\ell + 76\kappa$	$9\ell + 6\kappa$	✓	✓	✗
	Tetrad-R	2	Robust	2ℓ	3ℓ	2ℓ	3ℓ	✓	✓	✓

[a]Amortized costs are reported for 1 million operations [b]parties that carry out most of the computation during online phase [c]ℓ - size of ring in bits, κ - security parameter, d - length of the vectors. [d]A, B, G indicate support for arithmetic, boolean, and garbled worlds respectively [e]'Comm' - communication, 'pre' - preprocessing, 'on' - online [f]cf. §A-D for details

Table I: Comparison of actively-secure MPC frameworks (3PC and 4PC) for PPML.

Works such as [6], [9], [28] typically go for active security with abort, where the adversary can act maliciously to obtain the output and make honest parties abort. The stronger notion of fairness guarantees that either all or none of the parties obtain the output. This incentivizes the adversary to behave honestly in resource-expensive tasks such as PPML, as causing an abort will waste its resources. Trident [4] showed that fairness can be achieved at the cost of security with abort. In cases where the risk of failure of the system is too high, for instance, when deploying PPML for healthcare applications, participants might want to avoid the case when none of them receive the output. The way to tackle this issue is to modify protocols to guarantee that the correct output is always delivered to the participants irrespective of an adversary's misbehaviour. This is provided by guaranteed output delivery (GOD) or robustness. A robust protocol prevents the adversary from repeatedly causing the computations to rerun, thereby upholding the trust in the system. We propose two variants of the framework – one with fairness and the other with robustness. We detail the related work in §A and continue with our contributions.

A. Our Contributions

We make several contributions towards designing a practically efficient 4PC mixed-protocol framework, tolerating at most one active corruption. It operates over the ring $\mathbb{Z}_{2^\ell}$ and provides *end-to-end* conversions to switch between arithmetic, boolean and garbled worlds. We assume a one-time key setup phase and work in the (function-dependent) preprocessing model which paves the way for a fast online phase.

Depending on the sensitivity of the application and the underlying data, one might want different levels of security. For this, we propose two variants of the framework, covering fairness (Tetrad) and robustness (Tetrad-R) guarantees. The fair variant improves upon the state-of-the-art *fair* framework of Trident [4]. Tetrad-R improves communication over the best robust protocols [14], [29], while offering support for secure training of neural networks, which was not supported in previous works.

1) Improved Arithmetic/Boolean 4PC: In Tetrad, the multiplication protocol has a communication cost of only 5 ring elements as opposed to 6 in the state-of-the-art framework of Trident [4]. Security is elevated to robustness via Tetrad-R, which has a minimal overhead over the fair one, in the preprocessing. Concretely, for a 64-bit ring with 40-bit statistical security, the overhead per multiplication is 0.027 bits for a circuit containing 2^{20} multiplications. This means robustness essentially comes free in the case of large circuits.

A notable contribution is the design of the multiplication protocol. It gives the following benefits – i) support for on-demand applications, ii) probabilistic truncation without overhead and iii) multi-input multiplication gates.

On-demand applications: The design of the multiplication protocol allows Tetrad to support on-demand applications where a preprocessing phase is not available. This variant of the protocols (cf. §B) has a round complexity that is the same as that of the online phases of the protocols in the preprocessing model and retains the same overall communication. It takes advantage of parallelization, which is often not possible in the *function-dependent* preprocessing model where the preprocessing and the online phases must be executed sequentially.

Probabilistic truncation without any overhead: Multiplication (and dot product) with truncation forms an essential component while working with fixed-point values. Techniques for probabilistic truncation were proposed by [6], [7]. Recently, [28] gave an efficient instantiation of truncation for 4PC with abort, based on the technique of ABY3. Using that as a baseline, we demonstrate for the *first time* in the fair and robust settings, how multiplication (and dot-product) with truncation can be performed without any additional cost over a multiplication.

Multi-input multiplication: Inspired by [23], [30], we propose new protocols for 3 and 4-input multiplication, allowing multiplication of 3 and 4 inputs in one online round. Naively, performing a 4-input multiplication follows a tree-based approach, and the required communication is that of three 2-input multiplications and 2 online rounds.

Our contribution lies in keeping the communication and the round of the online phase the same as that of 2-input multiplication (i.e. invariant of the number of inputs). To achieve this, we trade off the preprocessing cost. Looking ahead, multi-input multiplication, when coupled with the optimized parallel prefix adder circuit from [23], brings in a $2\times$ improvement in online rounds. It also cuts down the online communication of secure comparison, impacting PPML applications.

2) 4PC Mixed-Protocol Framework: In addition to relying on the improved arithmetic/boolean world, we observe that a large portion of the computation in most MPC-based PPML frameworks is done over the arithmetic and boolean worlds. The garbled world is used only to perform the non-linear operations (e.g. softmax) that are expensive in the arithmetic/boolean world and switch back immediately after. Leveraging this observation we propose tailor-made GC-based protocols with *end-to-end* conversion techniques.

The tailor-made GC for the fair protocols, has the following advantages over Trident – i) no use of commitments for the inputs, and ii) no requirement of an explicit input sharing and output reconstruction phase, as explained later. The overall communication cost remains the same as Trident with 1 GC and 2 online rounds. In addition, for time-constrained applications we offer a variant that trades off 1 GC at the expense of 1 lesser online round. When it comes to robustness, the state-of-the-art for GC protocols are [31], costing 12 GC and 2 rounds, and [24], costing 2 GC and 4 rounds. We propose robust GC conversions for the first time, and they cost 2 GC and have an amortized round complexity of 1.

As mentioned earlier, the framework operates over three domains - arithmetic, boolean, and garbled (§IV). For an operation that required computing over the garbled domain, the standard approach is to first switch from *Arithmetic to Garbled* and evaluate the garbled circuit to obtain a garbled-shared output. These shares are brought back to the arithmetic domain using a *Garbled to Arithmetic* conversion. Our approach instead is to modify the garbled circuit such that the output is in the arithmetic domain. This eliminates the need for an explicit *Garbled to Arithmetic* conversion, saving in both communication and rounds in the online phase. More generally, end-to-end conversions are of the form "x-Garbled-x" where x can be either arithmetic or boolean, and need a single round for the garbled world (cf. §IV).

Comparison of Tetrad with actively secure PPML frameworks in 3PC and 4PC is presented in Table I. The dot product is chosen as a parameter as it is one of the most crucial building blocks in PPML applications.

3) Benchmarking and PPML: We demonstrate the practicality of the framework, which combines the arithmetic, boolean, garbled worlds via benchmarking. The training and inference phases of deep neural networks such as LeNet [32] and VGG16 [33] and the inference phase of Support Vector Machines are benchmarked.

The implementation section is presented through the lens of deployment scenarios with two different goals. Participants in the first scenario are interested in the shortest online runtime for the computation, whereas participants in the second one want to minimize the deployment cost. Correspondingly, there are variants of our framework that cater to both scenarios.

Considering online runtime as the metric, Tetrad_T is the time-optimized (T) variant with the fastest online phase. Tetrad_C is the cost-optimized (C) variant, minimizing deployment cost. This is measured via *monetary cost* [34], which helps to capture the effect of the total runtime of the parties, and communication together. Both variants are compared against Trident [4], and their relative performance is indicated in Table II. The comparison is with respect to run time, communication, monetary cost, and throughput (Table V).

Protocol	Training & Inference[a]			Training	Inference
	Time_on[b]	Com_tot	CT_tot	Cost	TP_on
Tetrad_T	●	◐	●	◐	●
Tetrad_C	◐	●	◐	●	◐
Trident	○	○	○	○	○

[a] 'Com' - Communication, 'Time' - Runtime, 'CT' - Cumulative Runtime, 'Cost' - Monetary Cost, 'TP_on' - Online Throughput, on - online, tot - total

[b] ○ - good, ◐ - better, ● - best, (w.r.t parameter considered)

Table II: Comparison of Trident [4] with the versions of Tetrad for deep neural networks (cf. NN-4 in §VI).

Trident requires 3 parties to be active for most of the online phase, the 4th party coming in only towards the end of the computation. In Tetrad, it is brought down to 2, having a significant impact on the monetary cost.

Table II shows that Tetrad is better when compared to Trident across all the parameters considered. Within Tetrad, Tetrad_T fares better when it comes to online run time for both training and inference, while Tetrad_C does better in terms of communication. When it comes to inference, throughput is more relevant than the cost, and here, the time-optimized variant fares the best. Robust variants follow the same trends, and the reasons behind them are elaborated in §VI.

II. Preliminaries and Definitions

We consider 4 parties denoted by $\mathcal{P} = \{P_0, P_1, P_2, P_3\}$ that are connected by pair-wise private and authentic channels in a synchronous network, and a static, active adversary that can corrupt at most 1 party. In the secure outsourced computation (SOC) setting, the 4 servers hired to carry out the computation enact the role of the 4 parties mentioned above. In this setting inputs, intermediate values, and outputs exist in a secret-shared form. For ML training, data owners secret-share their data to the servers, which train the model using MPC. The trained model can then be reconstructed towards the data owners. Our framework is secure even if the corrupt server colludes with an arbitrary number of data owners. For ML inference, the model owner secret-shares a pre-trained model among the servers. A client secret-shares its query amongst the servers, who carry out the inference via MPC. The output is reconstructed towards the client. Security is guaranteed against a corrupt server that colludes either with the model owner or with the client. We do not guarantee the privacy of the training data against attacks such as attribute inference, membership inference, or model inversion [35]–[37]. This is an orthogonal problem, and we consider it as out-of-scope of this work.

In Tetrad, parties rely on a one-time shared key setup (cf. §A for the ideal functionality) [2]–[4], [6], [8] to facilitate generation of correlated randomness non-interactively. Our protocols work over the arithmetic ring $\mathbb{Z}_{2^\ell}$ or boolean ring $\mathbb{Z}_{2^1}$. We use fixed-point arithmetic (FPA) [2]–[4], [6], [8] representation to deal with floating-point values where a decimal value is represented as an ℓ-bit integer in signed 2's complement representation. The most significant bit (MSB) represents the sign bit and x least significant bits are reserved for the fractional part. The ℓ-bit integer is then treated as an element of $\mathbb{Z}_{2^\ell}$ and operations are performed modulo 2^ℓ. We set $\ell = 64$, $x = 13$, with $\ell - x - 1$ bits for the integral part.

Notation II.1. *For a vector $\vec{\mathbf{a}}$, a_i denotes the i^{th} element in the vector. For two vectors $\vec{\mathbf{a}}$ and $\vec{\mathbf{b}}$ of length d, the dot product is given by, $\vec{\mathbf{a}} \odot \vec{\mathbf{b}} = \sum_{i=1}^{\mathsf{d}} \mathsf{a}_i \mathsf{b}_i$. Given two matrices $\mathbf{A}, \mathbf{B}$, the operation $\mathbf{A} \circ \mathbf{B}$ denotes the matrix multiplication.*

Notation II.2. *For a bit $\mathsf{b} \in \{0, 1\}$, b^{R} denotes the representation of the bit value b over the arithmetic ring $\mathbb{Z}_{2^\ell}$. In detail, all the bits of b^{R} will be zero except for the least significant bit, which is set to b.*

Primitives: For our constructs we use two standard primitives (cf. §A) (a) a *collision-resistant* hash function, denoted as $\mathsf{H}(\cdot)$; (b) a garbling scheme $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$.

a) Sharing Semantics: To enforce security, we perform computation on secret-shared data. For the arithmetic and boolean sharing, we follow a $(4, 1)$ replicated secret sharing (RSS) [4], denoted by $[\![\cdot]\!]$. To leverage the benefits of the preprocessing paradigm, we associate meaning to the shares and demarcate the parties in terms of their roles. Three of the shares of a $(4, 1)$ RSS for a value v can be generated in the preprocessing phase independent of the value to be shared, and their sum can be interpreted as a mask. The fourth share, dependent on v, can be computed in the online phase and can be treated as the masked value. We denote the three preprocessed shares as $\lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^2, \lambda_{\mathsf{v}}^3$ and the mask as $\lambda_{\mathsf{v}} = \lambda_{\mathsf{v}}^1 + \lambda_{\mathsf{v}}^2 + \lambda_{\mathsf{v}}^3$. The masked value is denoted as m_{v}, and $\mathsf{m}_{\mathsf{v}} = \mathsf{v} + \lambda_{\mathsf{v}}$.

Type	P_0	P_1	P_2	P_3
$[\cdot]$-sharing[a]	–	v^1	v^2	–
$(\!(\cdot)\!)$-sharing	–	v^1	v^2	v^3
$\langle\cdot\rangle$-sharing	–	$(\mathsf{v}^1, \mathsf{v}^3)$	$(\mathsf{v}^2, \mathsf{v}^3)$	$(\mathsf{v}^1, \mathsf{v}^2)$
$[\![\cdot]\!]$-sharing[b]	$(\lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^2, \lambda_{\mathsf{v}}^3)$	$(\mathsf{m}_{\mathsf{v}}, \lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^3)$	$(\mathsf{m}_{\mathsf{v}}, \lambda_{\mathsf{v}}^2, \lambda_{\mathsf{v}}^3)$	$(\mathsf{m}_{\mathsf{v}}, \lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^2)$

[a] $\mathsf{v} = \mathsf{v}_1 + \mathsf{v}_2 \ (+\mathsf{v}_3)$ [b] $\mathsf{m}_{\mathsf{v}} = \mathsf{v} + \lambda_{\mathsf{v}}$

Table III: Sharing semantics for a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ in Tetrad. All the shares are ℓ-bit ring elements.

Next, we distinguish the four parties into two sets; the *eval* set $\mathcal{E} = \{P_1, P_2\}$ which is assigned the task of carrying out the computation, and is active throughout the online phase. The *helper* set $\mathcal{D} = \{P_0, P_3\}$ is used to assist $\mathcal{E}$ in verification, so it is only active towards the end of the computation. Complying with the roles and the RSS format, the distribution is done as follows: $P_0 : \{\lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^2, \lambda_{\mathsf{v}}^3\}, P_1 : \{\lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^3, \mathsf{m}_{\mathsf{v}}\}, P_2 : \{\lambda_{\mathsf{v}}^2, \lambda_{\mathsf{v}}^3, \mathsf{m}_{\mathsf{v}}\}$, and $P_3 : \{\lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^2, \mathsf{m}_{\mathsf{v}}\}$. The shares are distributed among $\mathcal{D}$ such that P_3 gets m_{v} whereas P_0 gets all the shares of λ_{v}. During preprocessing, P_0 computes a part of the data needed for verification (cf. Fig. 3) using its input independent shares, which is communicated to P_3. This enables a verification in the online without P_0, for the fair protocols.

Exploiting the asymmetry of the roles allows for minimal online participation, giving a huge improvement in the cumulative runtime (sum of uptime of all the parties), thereby saving in monetary costs (cf. §VI). The RSS sharing semantics are presented in Table III, denoted by $[\![\cdot]\!]$, in a modular way with the help of three intermediate sharing semantics $[\cdot]$, $(\!(\cdot)\!)$ and $\langle\cdot\rangle$. All the sharing schemes used are linear i.e. given shares of values $\mathsf{v}_1, \ldots, \mathsf{v}_m$ and public constants $c_1, \ldots, c_m$, sharing of $\sum_{i=1}^{m} c_i \mathsf{v}_i$ can be computed locally for an integer m.

Notation II.3. *(a) For the $[\![\cdot]\!]$-shares of n values $\mathsf{a}_1, \ldots, \mathsf{a}_n$, $\gamma_{\mathsf{a}_1 \ldots \mathsf{a}_n} = \prod_{i=1}^{n} \lambda_{\mathsf{a}_i}$ and $\mathsf{m}_{\mathsf{a}_1 \ldots \mathsf{a}_n} = \prod_{i=1}^{n} \mathsf{m}_{\mathsf{a}_i}$ (b) We use superscripts $\mathbf{B}$, and $\mathbf{G}$ to denote sharing semantics in boolean, and garbled world, respectively– $[\![\cdot]\!]^{\mathbf{B}}$, $[\![\cdot]\!]^{\mathbf{G}}$. We omit the superscript for arithmetic world.*

Sharing semantics for boolean sharing over $\mathbb{Z}_2$ is similar to arithmetic sharing except that addition is replaced with XOR. The semantics for garbled sharing are described in §IV with the relevant context.

III. 4PC Protocol

This section covers the details of our 4PC protocol over an arithmetic ring $\mathbb{Z}_{2^\ell}$. We begin by explaining the relevant primitives in §III-A. The multiplication protocol with abort is presented in §III-B, followed by details on elevating the security to fairness in §III-C. Lastly, in §III-D, we show how to improve the security to robustness[1]. Formal details along with the cost analysis for the protocols is deferred to §B.

A. Primitives

a) Joint-Send (jsnd): The Joint-Send (jsnd) primitive allows two parties P_i, P_j to relay a message v to a third party P_k ensuring either the delivery of the message or `abort` in case of inconsistency. Towards this, P_i sends v to P_k, while P_j sends a hash of the same, $\mathsf{H}(\mathsf{v})$, to P_k. Party P_k accepts the message if the hash values are consistent and `aborts` otherwise. Note that the communication of the hash can be clubbed together for several instances and be deferred to the end of the protocol, amortizing the cost.

b) Joint-Send (jsnd) for robust protocols: To achieve robustness, we instantiate jsnd using the joint-message passing (jmp) primitive of [14]. The jsnd primitive (Fig. 12) allows two senders P_i, P_j to relay a common message, $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, to a recipient P_k, either by ensuring successful delivery of v, or by establishing a conflicting pair of parties, one among which is guaranteed to be corrupt. This implies the residual two parties are honest, one of which is then entrusted to take the computation to completion by enacting the role of a trusted party (P_{TP}). The instantiation of jsnd can be viewed as consisting of two phases (*send, verify*), where the *send* phase consists of P_i sending v to P_k and the rest of the protocol steps go to *verify* phase (which ensures correct *send* or P_{TP} identification). This requires 1 round of interaction and ℓ bits of communication. To leverage amortization, *verify* will be executed only once, at the end of the computation, and requires 2 rounds.

[1]The classical notion of robustness is achieved

The jsnd primitive is instantiated depending on the desired security guarantee. For simplicity, we give common constructions for fair and robust variants of the protocols, when they only differ in the instantiation of jsnd.

Notation III.1. *Protocol* Π_{jsnd} *denotes the instantiation of Joint-Send* (jsnd) *primitive. We say that* P_i, P_j jsnd v *to* P_k *when they invoke* $\Pi_{\mathsf{jsnd}}(P_i, P_j, \mathsf{v}, P_k)$.

c) Sharing: Protocol Π_{Sh} (Fig. 1) enables P_i to generate $[\![\cdot]\!]$-share of a value v. During the preprocessing phase, λ-shares are sampled non-interactively using the pre-shared keys (cf. §A-B0a) in a way that P_i will get the entire mask λ_{v}. During the online phase, P_i computes $\mathsf{m_v} = \mathsf{v} + \lambda_{\mathsf{v}}$ and sends to P_1, P_2, P_3, which exchange the hash values to check for consistency. Parties abort in the fair protocol in case of inconsistency, whereas for robust security, parties proceed with a default value.

Protocol $\Pi_{\mathsf{Sh}}(P_i, \mathsf{v})$

Preprocessing: Sample the following:

$P_i, P_0, P_1, P_3 : \lambda_{\mathsf{v}}^1$ | $P_i, P_0, P_2, P_3 : \lambda_{\mathsf{v}}^2$ | $P_i, P_0, P_1, P_2 : \lambda_{\mathsf{v}}^3$

Online:

1) P_i computes $\mathsf{m_v} = \mathsf{v} + \lambda_{\mathsf{v}}$ and sends to P_1, P_2, P_3.
2) P_1, P_2, P_3 mutually exchange $\mathsf{H}(\mathsf{m_v})$ and accept the sharing if there exists a majority. Else parties `abort` for the case of fairness and accept a default value for the case of robust security.

Figure 1: $[\![\cdot]\!]$-sharing of a value v by party P_i.

d) Joint Sharing: Protocol Π_{JSh} enables parties P_i, P_j to generate $[\![\cdot]\!]$-share of a value v. The protocol is similar to Π_{Sh} except that P_j ensures the correctness of the sharing performed by P_i. During the preprocessing, λ-shares are sampled such that both P_i, P_j will get the entire mask λ_{v}. During the online phase, P_i, P_j compute and jsnd $\mathsf{m_v} = \mathsf{v} + \lambda_{\mathsf{v}}$ to parties P_1, P_2, P_3.

For joint-sharing a value v possessed by P_0 along with another party in the preprocessing, the communication can be optimized further. The protocol steps based on the (P_i, P_j) pair are summarised below:

- $(P_0, P_1) : \mathcal{P} \setminus \{P_2\}$ sample $\lambda_{\mathsf{v}}^1 \in_R \mathbb{Z}_{2^\ell}$; Set $\lambda_{\mathsf{v}}^2 = \mathsf{m_v} = 0$; P_0, P_1 jsnd $\lambda_{\mathsf{v}}^3 = -\mathsf{v} - \lambda_{\mathsf{v}}^1$ to P_2.
- $(P_0, P_2) : \mathcal{P} \setminus \{P_3\}$ sample $\lambda_{\mathsf{v}}^3 \in_R \mathbb{Z}_{2^\ell}$; Set $\lambda_{\mathsf{v}}^1 = \mathsf{m_v} = 0$; P_0, P_2 jsnd $\lambda_{\mathsf{v}}^2 = -\mathsf{v} - \lambda_{\mathsf{v}}^3$ to P_3.
- $(P_0, P_3) : \mathcal{P} \setminus \{P_1\}$ sample $\lambda_{\mathsf{v}}^2 \in_R \mathbb{Z}_{2^\ell}$; Set $\lambda_{\mathsf{v}}^3 = \mathsf{m_v} = 0$; P_0, P_3 jsnd $\lambda_{\mathsf{v}}^1 = -\mathsf{v} - \lambda_{\mathsf{v}}^1$ to P_1.

e) Reconstruction: Protocol $\Pi_{\mathsf{Rec}}(\mathcal{P}, \mathsf{v})$ (Fig. 13) enables parties in $\mathcal{P}$ to compute v, given its $[\![\cdot]\!]$-share. Note that each party misses one share to reconstruct the output, and the other 3 parties hold this share. 2 out of the 3 parties will jsnd the missing share to the party that lacks it. Reconstruction towards a single party can be viewed as a special case.

f) $\mathcal{F}_{\mathsf{zero}}$ *- Generating additive shares of zero:* In Tetrad, we make use of a functionality $\mathcal{F}_{\mathsf{zero}}$ to enable parties P_0, P_i obtain Z_i for $i \in \{1, 2, 3\}$ such that $Z_1 + Z_2 + Z_3 = 0$. We observe that the functionality can be instantiated non-interactively using the pre-shared keys (cf. §A-B0a). For this, parties in $\mathcal{P} \setminus \{P_j\}$ sample random value r_j for $j \in \{1, 2, 3\}$. The shares are then defined as $Z_1 = \mathsf{r}_3 - \mathsf{r}_2, Z_2 = \mathsf{r}_1 - \mathsf{r}_3$ and $Z_3 = \mathsf{r}_2 - \mathsf{r}_1$.

g) Multiplication of $\langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle$*, held in clear by* P_0*:* To multiply $\langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle$, where $\mathsf{a}, \mathsf{b} \in \mathbb{Z}_{2^\ell}$ are held in clear by P_0, and generate $\langle \mathsf{z} \rangle$ such that $\mathsf{z} = \mathsf{ab}$, Π_{MulR} (Fig. 2) proceed as follows. Parties locally generate a $(\!(\cdot)\!)$-sharing of z, where P_0 knows all three $(\!(\cdot)\!)$-shares. To complete the generation of $\langle \mathsf{z} \rangle$, P_0, P_i for $i \in \{1, 2, 3\}$, randomize their $(\!(\cdot)\!)$-share of z using $(\!(\cdot)\!)$-share of 0, and jsnd $(\!(\mathsf{z})\!)^i$, to one other party.

Protocol $\Pi_{\mathsf{MulR}}(\langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle)$

1) Invoke $\mathcal{F}_{\mathsf{zero}}$ to enable P_0, P_j obtain Z_j for $j \in \{1, 2, 3\}$ such that $Z_1 + Z_2 + Z_3 = 0$.

$$P_0, P_1 \text{ jsnd } (\!(\mathsf{z})\!)^1 = \mathsf{a}^1\mathsf{b}^3 + \mathsf{a}^3\mathsf{b}^1 + \mathsf{a}^3\mathsf{b}^3 + Z_1 \text{ to } P_2.$$
$$P_0, P_2 \text{ jsnd } (\!(\mathsf{z})\!)^2 = \mathsf{a}^2\mathsf{b}^3 + \mathsf{a}^3\mathsf{b}^2 + \mathsf{a}^2\mathsf{b}^2 + Z_2 \text{ to } P_3.$$
$$P_0, P_3 \text{ jsnd } (\!(\mathsf{z})\!)^3 = \mathsf{a}^1\mathsf{b}^2 + \mathsf{a}^2\mathsf{b}^1 + \mathsf{a}^1\mathsf{b}^1 + Z_3 \text{ to } P_1.$$

2) Set $\langle \mathsf{z} \rangle$ as $\mathsf{z}^1 = (\!(\mathsf{z})\!)^3, \quad \mathsf{z}^2 = (\!(\mathsf{z})\!)^2, \quad \mathsf{z}^3 = (\!(\mathsf{z})\!)^1$.

Figure 2: Multiplication of $\langle \cdot \rangle$-shared values, held on clear by P_0.

B. Multiplication in Tetrad

Given the shares of a, b, the goal of the multiplication protocol is to generate shares of $\mathsf{z} = \mathsf{ab}$. The protocol is designed such that parties P_1, P_2 obtain a masked version of the output z, say $\mathsf{z} - \mathsf{r}$ in the online phase, and P_0, P_3 obtain the mask r in the preprocessing phase. Parties then generate $[\![\cdot]\!]$-sharing of these values by executing Π_{JSh}, and locally compute $[\![\mathsf{z} - \mathsf{r}]\!] + [\![\mathsf{r}]\!]$ to obtain the final output.

a) Online: Note that,

$$\begin{aligned} \mathsf{z} - \mathsf{r} &= \mathsf{ab} - \mathsf{r} = (\mathsf{m_a} - \lambda_{\mathsf{a}})(\mathsf{m_b} - \lambda_{\mathsf{b}}) - \mathsf{r} \\ &= \mathsf{m_{ab}} - \mathsf{m_a}\lambda_{\mathsf{b}} - \mathsf{m_b}\lambda_{\mathsf{a}} + \gamma_{\mathsf{ab}} - \mathsf{r} \quad \text{(cf. notation II.3)} \end{aligned} \tag{1}$$

In Eq 1, P_1, P_2 can compute $\mathsf{m_{ab}}$ locally, and hence we are interested in computing $\mathsf{y} = (\mathsf{z} - \mathsf{r}) - \mathsf{m_{ab}}$. Let us view y as $\mathsf{y} = \mathsf{y}_1 + \mathsf{y}_2 + \mathsf{y}_3$, where y_1 and y_2 can be computed respectively by P_1 and P_2, and y_3 consists of terms that can be computed by both.

$$\begin{aligned} P_1 &: \mathsf{y}_1 = -\lambda_{\mathsf{a}}^1\mathsf{m_b} - \lambda_{\mathsf{b}}^1\mathsf{m_a} + [\gamma_{\mathsf{ab}} - \mathsf{r}]_1 \\ P_2 &: \mathsf{y}_2 = -\lambda_{\mathsf{a}}^2\mathsf{m_b} - \lambda_{\mathsf{b}}^2\mathsf{m_a} + [\gamma_{\mathsf{ab}} - \mathsf{r}]_2 \\ P_1, P_2 &: \mathsf{y}_3 = -\lambda_{\mathsf{a}}^3\mathsf{m_b} - \lambda_{\mathsf{b}}^3\mathsf{m_a} \end{aligned} \tag{2}$$

The preprocessing is set up such that P_1, P_2 receive additive shares ($[\cdot]$) of $\gamma_{\mathsf{ab}} - \mathsf{r}$. P_1, P_2 then mutually exchange the missing share to reconstruct y and subsequently $\mathsf{z} - \mathsf{r}$.

b) Verification: To ensure correctness of the values exchanged in the online phase, we use the assistance of P_3. Concretely, P_3 obtains $\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{s}$, where s is a random mask known to P_0, P_1, P_2. For this, P_3 needs $\gamma_{\mathsf{ab}} + \mathsf{s}$, which it obtains from the preprocessing phase. The mask s is used to prevent the leakage from γ_{ab} to P_3. P_3 computes a hash of $\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{s}$ and sends it to P_1, P_2, which `abort` if it is inconsistent.

c) Preprocessing: Parties should obtain the following values from the preprocessing phase:

i) $P_1, P_2 : [\gamma_{\mathsf{ab}} - \mathsf{r}]$ | ii) $P_0, P_3 : \mathsf{r}$ | iii) $P_3 : \gamma_{\mathsf{ab}} + \mathsf{s}$

For i) and ii), let $\gamma_{\mathsf{ab}} = \gamma^1_{\mathsf{ab}} + \gamma^2_{\mathsf{ab}} + \gamma^3_{\mathsf{ab}}$, where P_0 along with P_i can compute γ^i_{ab} for $i \in \{1, 2, 3\}$. For P_1, P_2, to form an additive sharing of $(\gamma_{\mathsf{ab}} - r)$, it suffices for them to define their share as $\gamma^i_{\mathsf{ab}} + \left[\gamma^3_{\mathsf{ab}} - r\right]$. Instead of sampling a fresh random value for r, P_0, P_3, along with P_i, sample the share for $\gamma^3_{\mathsf{ab}} - \mathsf{r}$ as u^i for $i \in \{1, 2\}$. P_0, P_3 compute r as $\gamma^3_{\mathsf{ab}} - \mathsf{u}^1 - \mathsf{u}^2$. Note that r computed this way is still uniformly random, as $\mathsf{u}^1, \mathsf{u}^2$ are sampled uniformly at random.

For iii), P_3 needs $\mathsf{w} = \gamma^1_{\mathsf{ab}} + \gamma^2_{\mathsf{ab}} + \mathsf{s}$. To tackle this, P_0, P_1, P_2 sample $\mathsf{s}_1, \mathsf{s}_2$, and set $\mathsf{s} = \mathsf{s}_1 + \mathsf{s}_2$. P_0, P_i, for $i \in \{1, 2\}$, jsnd $\gamma^i_{\mathsf{ab}} + \mathsf{s}_i$ to P_3. This requires a communication of 2 elements. As an optimization, P_0 sends w to P_3. If P_0 is malicious, it might send a wrong value to P_3. However, in this case, every party in the online phase would be honest. And since P_1, P_2 do not use w in their computation, any error in w is bound to get caught in the verification phase.

Protocol $\Pi_{\mathsf{Mult}}(\mathsf{a}, \mathsf{b}, \mathsf{isTr})$

Let isTr be a bit that denotes whether truncation is required ($\mathsf{isTr} = 1$) or not ($\mathsf{isTr} = 0$).

Preprocessing:

1) Locally compute:

$$P_0, P_1 : \gamma^1_{\mathsf{ab}} = \lambda^1_{\mathsf{a}}\lambda^3_{\mathsf{b}} + \lambda^3_{\mathsf{a}}\lambda^1_{\mathsf{b}} + \lambda^3_{\mathsf{a}}\lambda^3_{\mathsf{b}}$$
$$P_0, P_2 : \gamma^2_{\mathsf{ab}} = \lambda^2_{\mathsf{a}}\lambda^3_{\mathsf{b}} + \lambda^3_{\mathsf{a}}\lambda^2_{\mathsf{b}} + \lambda^2_{\mathsf{a}}\lambda^2_{\mathsf{b}}$$
$$P_0, P_3 : \gamma^3_{\mathsf{ab}} = \lambda^1_{\mathsf{a}}\lambda^2_{\mathsf{b}} + \lambda^2_{\mathsf{a}}\lambda^1_{\mathsf{b}} + \lambda^1_{\mathsf{a}}\lambda^1_{\mathsf{b}}$$

2) P_0, P_3 and P_j sample random $\mathsf{u}^j \in_R \mathbb{Z}_{2^\ell}$ for $j \in \{1, 2\}$. Let $\mathsf{u}^1 + \mathsf{u}^2 = \gamma^3_{\mathsf{ab}} - \mathsf{r}$ for a random $\mathsf{r} \in_R \mathbb{Z}_{2^\ell}$.
3) P_0, P_3 compute $\mathsf{r} = \gamma^3_{\mathsf{ab}} - \mathsf{u}^1 - \mathsf{u}^2$ and set $\mathsf{q} = \mathsf{r}^\mathsf{t}$ if $\mathsf{isTr} = 1$, else set $\mathsf{q} = \mathsf{r}$. P_0, P_3 execute $\Pi_{\mathsf{JSh}}(P_0, P_3, \mathsf{q})$ to generate $[\![\mathsf{q}]\!]$.
4) P_0, P_1, P_2 sample random $\mathsf{s}_1, \mathsf{s}_2 \in_R \mathbb{Z}_{2^\ell}$ and set $\mathsf{s} = \mathsf{s}_1 + \mathsf{s}_2$[a]. P_0 sends $\mathsf{w} = \gamma^1_{\mathsf{ab}} + \gamma^2_{\mathsf{ab}} + \mathsf{s}$ to P_3.

Online: Let $\mathsf{y} = (\mathsf{z} - \mathsf{r}) - \mathsf{m_a m_b}$.

1) Locally compute:

$$P_1 : \mathsf{y}_1 = -\lambda^1_{\mathsf{a}}\mathsf{m_b} - \lambda^1_{\mathsf{b}}\mathsf{m_a} + \gamma^1_{\mathsf{ab}} + \mathsf{u}^1$$
$$P_2 : \mathsf{y}_2 = -\lambda^2_{\mathsf{a}}\mathsf{m_b} - \lambda^2_{\mathsf{b}}\mathsf{m_a} + \gamma^2_{\mathsf{ab}} + \mathsf{u}^2$$
$$P_1, P_2 : \mathsf{y}_3 = -\lambda^3_{\mathsf{a}}\mathsf{m_b} - \lambda^3_{\mathsf{b}}\mathsf{m_a}$$

2) P_1 sends y_1 to P_2, while P_2 sends y_2 to P_1, and they locally compute $\mathsf{z} - \mathsf{r} = (\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{y}_3) + \mathsf{m_a m_b}$.
3) If $\mathsf{isTr} = 1$, P_1, P_2 set $\mathsf{p} = (\mathsf{z} - \mathsf{r})^\mathsf{t}$, else $\mathsf{p} = \mathsf{z} - \mathsf{r}$. P_1, P_2 execute $\Pi_{\mathsf{JSh}}(P_1, P_2, \mathsf{p})$ to generate $[\![\mathsf{p}]\!]$.
4) Locally compute $[\![\mathsf{o}]\!] = [\![\mathsf{p}]\!] + [\![\mathsf{q}]\!]$. Here $\mathsf{o} = \mathsf{z}^\mathsf{t}$ if $\mathsf{isTr} = 1$ and z otherwise.
5) *Verification:* P_3 computes $\mathsf{v} = -(\lambda^1_{\mathsf{a}} + \lambda^2_{\mathsf{a}})\mathsf{m_b} - (\lambda^1_{\mathsf{b}} + \lambda^2_{\mathsf{b}})\mathsf{m_a} + \mathsf{u}^1 + \mathsf{u}^2 + \mathsf{w}$ and sends $\mathsf{H}(\mathsf{v})$ to P_1 and P_2. Parties P_1, P_2 `abort` iff $\mathsf{H}(\mathsf{v}) \neq \mathsf{H}(\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{s})$.

[a] For the fair protocol, it is enough for P_0, P_1, P_2 to sample s directly.

Figure 3: Multiplication with / without truncation in Tetrad.

d) Truncation: For a value $\mathsf{v} = \mathsf{v}_1 + \mathsf{v}_2$, SecureML [7] showed that the truncated value $\mathsf{v}/2^x$, denoted by v^t, can be computed as $\mathsf{v}^\mathsf{t}_1 + \mathsf{v}^\mathsf{t}_2$. With high probability, a truncated value having at most one bit error in the least significant position is generated. It was shown in SecureML that accuracy drop for ML algorithms due to the one bit error is minimal. However, the method cannot be generalized to more than two parties. ABY3 [6] demonstrated the extension to 3-party setting with a generic design that uses a truncation pair of the form $(\mathsf{r}, \mathsf{r}^\mathsf{t})$. Here, r is a random value and r^t denotes its truncated version. Given this pair, z can be truncated by opening $\mathsf{z} - \mathsf{r}$ towards all, and computing z^t as $\mathsf{z}^\mathsf{t} = (\mathsf{z} - \mathsf{r})^\mathsf{t} + \mathsf{r}^\mathsf{t}$. Note that all operations are carried out on shares.

The design of our multiplication allows for truncation to be carried out this way without any additional overhead in communication. Towards this, P_1, P_2 locally truncate $(\mathsf{z} - \mathsf{r})$ and generate $[\![\cdot]\!]$-shares of it in the online phase. Similarly, P_0, P_3 truncate r in the preprocessing phase and generate its $[\![\cdot]\!]$-shares. Then $[\![\mathsf{z}^\mathsf{t}]\!] = [\![(\mathsf{z} - \mathsf{r})^\mathsf{t}]\!] + [\![\mathsf{r}^\mathsf{t}]\!]$

e) Multiplication by constant: This operation in MPC is typically local: given constant α and $[\![\mathsf{v}]\!]$, the product can be written as $\alpha\mathsf{v} = \beta^1 + \beta^2$ where $\beta^1 = \alpha.(\mathsf{m_v} - \lambda^3_{\mathsf{v}})$ and $\beta^2 = \alpha.(-\lambda^1_{\mathsf{v}} - \lambda^2_{\mathsf{v}})$. However, in FPA, we need to perform a truncation on the output. For this P_1, P_2 truncate β^1 and execute Π_{JSh}, while P_0, P_3 do the same with β^2.

C. Achieving Fairness

Here we show how to extend the security of Tetrad from abort to fairness using techniques from Trident [4]. Before proceeding with the output reconstruction, we need to ensure that all the honest parties are alive after the verification phase. For this, all the parties maintain an *aliveness* bit, say b, which is initialized to `continue`. If the verification phase is not successful for a party, it sets $\mathsf{b} =$ `abort`. In the first round of reconstruction, the parties mutually exchange their b bit and accept the value that forms the majority. Since we have only one corruption, it is guaranteed that all the honest parties will be in agreement on b. If $\mathsf{b} =$ `continue`, then the parties exchange their missing shares and accept the majority. As per the sharing semantics, every missing share is possessed by three parties, out of which there can be at most one corruption. As an optimization, for instances where many values are reconstructed, two out of the three parties can send the share while the third can send a hash of the same.

D. Achieving Robustness

Here we show how to extend the security of Tetrad to provide robustness while retaining the same amortized communication complexity. The robust variant, denoted by Tetrad-R, additionally requires a verification check in the preprocessing phase of multiplication as compared to Tetrad. Moreover, the reconstruction protocol is similar to the fair counterpart, except that aliveness check is not required since a cheating would result in identifying an honest party ($\mathsf{P_{TP}}$).

The multiplication protocol Π_{Mult} (Fig. 3) is modified as follows. First, the robust variant of Π_{JSh} is used instead of the fair one. This ensures correctness of messages to be communicated or identifies a conflicting pair of parties, one among which is guaranteed to be corrupt. Next, to ensure the correctness of w sent by P_0 alone in the preprocessing phase, we introduce Π_{VrfyP0} (Fig. 4). If Π_{VrfyP0} fails, parties identify a $\mathsf{P_{TP}}$ in the preprocessing phase itself. Finally, in case of an `abort` in the online phase (which proceeds similar to the that of Π_{Mult}), P_0 is assigned as the $\mathsf{P_{TP}}$. Since P_0 does not participate in the online phase of multiplication, and its communication in the preprocessing has been verified via Π_{VrfyP0}, this assignment is safe.

Verifying the communication by P_0: In Π_{Mult} (Fig. 3), P_0 computes and sends $\mathsf{w} = \gamma^1_{\mathsf{ab}} + \gamma^2_{\mathsf{ab}} + \mathsf{s}_1 + \mathsf{s}_2$ to P_3, where P_0, P_1, P_2 know $\mathsf{s}_1, \mathsf{s}_2$ in clear. Note that $\mathsf{w} = \mathsf{w}^1 + \mathsf{w}^2$ for $\mathsf{w}^1 = \gamma^1_{\mathsf{ab}} + \mathsf{s}_1$ and $\mathsf{w}^2 = \gamma^2_{\mathsf{ab}} + \mathsf{s}_2$. Also, P_0 along with P_1, P_2 and P_3 possess the values $\mathsf{w}^1, \mathsf{w}^2$ and w respectively. Checking the correctness of w thus reduces to verifying if $\mathsf{w} = \mathsf{w}^1 + \mathsf{w}^2$.

To verify this relation for all M multiplication gates in the circuit, i.e. $\{\mathsf{w}_j \stackrel{?}{=} \mathsf{w}^1_j + \mathsf{w}^2_j\}_{j\in[M]}$, one approach is to compute a random linear combination and verify the relation on the sum. While working over a field $\mathbb{F}_p$, this solution has an error probability $1/|\mathbb{F}_p|$, where $|\mathbb{F}_p|$ denotes the size of $\mathbb{F}_p$. However, this solution does not work naively over rings since not every element in the ring has an inverse, as opposed to fields. Concretely, the check can still pass with a probability of at most $1/2$ [38], [39]. To reduce the cheating probability, the check is repeated κ times, thereby bounding the cheating probability by $1/2^\kappa$. As an optimization, it is sufficient to choose the random combiners from $\{0, 1\}$. Thus, for one check, parties need to sample only a binary string of M bits using the shared-key. The formal verification protocol appears in Fig. 4.

Protocol $\Pi_{\mathsf{VrfyP0}}(\{[\mathsf{w}_j]\}_{j=1}^M)$

Repeat the following κ times, in parallel.

1) Sample random values $\tau_1, \ldots, \tau_M \in \mathbb{Z}_{2^\ell}$.
2) Locally compute: $P_0, P_1 : \mathsf{e}^1 = \sum_{j=1}^M \tau_j \mathsf{w}^1_j$; $P_0, P_2 : \mathsf{e}^2 = \sum_{j=1}^M \tau_j \mathsf{w}^2_j$; $P_0, P_3 : \mathsf{e} = \sum_{j=1}^M \tau_j \mathsf{w}_j$.
3) (P_0, P_1), (P_0, P_2) and (P_0, P_3) generate $[\![\cdot]\!]$-shares of $\mathsf{e}^1, \mathsf{e}^2$ and e respectively using Π_{JSh}.
4) Locally compute $[\![\mathsf{g}]\!] = [\![\mathsf{e}]\!] - [\![\mathsf{e}^1]\!] - [\![\mathsf{e}^2]\!]$.
5) Robustly reconstruct g and check if $\mathsf{g} \stackrel{?}{=} 0$.

If for all κ repetitions, $\mathsf{g} = 0$, then continue with rest of the computation. Else, P_0 is identified to be corrupt and $\mathsf{P_{TP}} = P_1$.

Figure 4: Verification of P_0's communication in the multiplication protocol of Tetrad-R

The robust protocol can be optimized further if cheating is detected (`abort` signal is generated) in the preprocessing phase. Concretely, this can be identified in the preprocessing phase either from the verification of jsnd instances or output of Π_{VrfyP0}. When such a cheating is detected, the corrupt party is identified as follows. Parties first broadcast their shared keys established in the key-setup phase (cf. §A-B0a). They recompute all the preprocessing data and verify against the data that was communicated to identify the corrupt party. Note that disclosing the shared keys does not violate input privacy because the preprocessing data is input independent. On identifying the corrupt party, it is eliminated from the computation, and a semi-honest 3-party computation is performed from this point onwards.

E. The complete 4PC

The above primitives can be compiled to compute an arithmetic circuit over $\mathbb{Z}_{2^\ell}$ as follows.

Parties first invoke the key-setup functionality $\mathcal{F}_{\mathsf{setup}}$ (Fig. 9) for key distribution, and preprocessing of input sharing (Π_{Sh}) and multiplication (Π_{Mult}), as per the given circuit. This generates the masks (λ) for all the wires in the circuit as per the sharing semantics. The preprocessing for linear gates can be performed non-interactively. The verification of all the protocols is executed before moving on to the online phase.

During the online phase, $P_i \in \mathcal{P}$ shares its input $\mathsf{x_i}$ by executing online steps of Π_{Sh} (Fig. 1). Parties then evaluate the gates in the circuit in the topological order, with linear gates being computed locally, and multiplication gates being computed via online phase of Π_{Mult} (Fig. 3). Finally, Π_{Rec} (Fig. 13) is executed for the output wires to reconstruct the function output.

F. Supporting on-demand computations

For on-demand applications where the underlying function to be computed is not known in advance, the preprocessing model is not desirable. We observe that the Tetrad protocol can be modified by executing the preprocessing phase in the online phase itself, keeping the same overall communication cost. The formal protocol appears in Fig. 14.

IV. Mixed Protocol Framework

In the applications we consider, the garbled circuit is used as an intermediary to evaluate certain functions where the input to the function as well as the output are in $[\![\cdot]\!]$-shared (or $[\![\cdot]\!]^{\mathbf{B}}$-shared) form. For this, we design end-to-end conversions which are of the form "x-Garbled-x" where x can be either arithmetic or boolean.

Similar to Trident [4], we design a fair GC world, using techniques from [40], that requires communicating 1 GC and 2 rounds for end-to-end conversions. We further extend it to provide robustness without inflating the cost. Due to its close resemblance to Trident, the details are deferred to §D-C. We observe that the online rounds for end-to-end conversions can be further reduced to 1 at the expense of communicating one more GC in a parallel execution. Note that a similar approach of using 2 parallel executions in Trident does not lead to obtaining a 1-round conversion due to their protocol design and reliance on piece-wise conversions. A high-level comparison is provided in Table IV, and more details are deferred to §E.

When compared to the standalone protocol of [40], the customized fair GC protocol for mixed framework eliminates the need for commitments to ensure input consistency and explicit input sharing and output reconstruction phases. For robustness, the standalone GC protocols of [31] requires communicating 12 GCs in 2 rounds while [24] communicates 2 GCs in 4 rounds. On the other hand, the robust variant in this work requires communicating 2 GC in 1 round. Moreover, these protocols leverage the benefit of amortization which comes from using jsnd.

Protocol[a]	Reference	Communication[b] (Preprocessing)	Rounds (Online)	Communication (Online)
2 GC variant	Trident	$6\ell\kappa + \ell$	2	$4\ell\kappa + 2\ell$
	Tetrad		1	$4\ell\kappa + \ell$
1 GC variant	Trident	$3\ell\kappa + \ell$	2	$2\ell\kappa + 3\ell$
	Tetrad		2	$2\ell\kappa + 2\ell$

[a] Notations: ℓ - size of ring in bits, κ - computational security parameter.
[b] Cost of GC is omitted, see Tables XI, XII for more details.

Table IV: End-to-end conversions in Trident [4] and Tetrad.

Leveraging an honest majority among the garblers and using jsnd, we only need semi-honest GC computation to get active security. Moreover, the state-of-the-art GC optimizations of free-XOR [41], [42], half gates [43], [44], and fixed AES-key [45] are deployed in our protocol.

A. GC for mixed protocol framework

The 2 GC variant has two parallel executions, each comprising of 3 garblers and 1 evaluator. P_1, P_2 act as evaluators in two independent executions and the parties in $\Phi_1 = \{P_0, P_2, P_3\}$, $\Phi_2 = \{P_0, P_1, P_3\}$ act as garblers, respectively. Note that it suffices for only P_0, P_3 to generate and jsnd the GC to the evaluator.

Garbled evaluation proceeds in three phases– i) Input phase, ii) Evaluation, and iii) Output phase. The input phase involves transferring the keys to the evaluators for every input to the GC. Note here that the function (to be evaluated via the GC) input is already $[\![\cdot]\!]^{\mathbf{B}}$-shared. Since each share of the function input is available with two garblers in each garbling instance, the correct key transfer is ensured via jsnd. The evaluation consists of GC transfer followed by GC evaluation. Lastly, in the output phase, evaluators obtain the encoded output. Preliminary details about the garbling scheme and additional details of the GC protocol are given in §D.

a) Input Phase: Given that the function input x is already available as $[\![\mathsf{x}]\!]^{\mathbf{B}}$, the boolean values $\mathsf{m_x}, \alpha_\mathsf{x}, \lambda^3_\mathsf{x}$, where $\alpha_\mathsf{x} = \lambda^1_\mathsf{x} \oplus \lambda^2_\mathsf{x}$ and $\mathsf{x} = \mathsf{m_x} \oplus \alpha_\mathsf{x} \oplus \lambda^3_\mathsf{x}$, act as the *new* inputs for the garbled computation, and garbled sharing ($[\![\cdot]\!]^{\mathbf{G}}$) is generated for each of these values. The semantics of $[\![\cdot]\!]^{\mathbf{B}}$-sharing ensures that each of these shares $(\mathsf{m_x}, \alpha_\mathsf{x}, \lambda^3_\mathsf{x})$ is available with two garblers in each garbling instance. The keys for the shares can either be sent (using jsnd) correctly to the evaluators or the inconsistency is detected. This key delivery essentially generates $[\![\cdot]\!]^{\mathbf{G}}$-sharing for each of these three values which enables GC evaluation. Thus, the goal of our input phase is to create the compound sharing, $[\![\mathsf{x}]\!]^{\mathbf{C}} = ([\![\mathsf{m_x}]\!]^{\mathbf{G}}, [\![\alpha_\mathsf{x}]\!]^{\mathbf{G}}, [\![\lambda^3_\mathsf{x}]\!]^{\mathbf{G}})$ for every input x to the function to be evaluated via the GC. We first discuss the semantics for $[\![\cdot]\!]^{\mathbf{G}}$-sharing followed by steps for generating $[\![\cdot]\!]^{\mathbf{C}}$-sharing.

b) Garbled sharing semantics: A value $\mathsf{v} \in \mathbb{Z}_2$ is $[\![\cdot]\!]^{\mathbf{G}}$-shared (garbled shared) amongst $\mathcal{P}$ if $P_i \in \{P_0, P_3\}$ holds $[\![\mathsf{v}]\!]^{\mathbf{G}}_i = (\mathsf{K}^{0,1}_\mathsf{v}, \mathsf{K}^{0,2}_\mathsf{v})$, P_1 holds $[\![\mathsf{v}]\!]^{\mathbf{G}}_1 = (\mathsf{K}^{\mathsf{v},1}_\mathsf{v}, \mathsf{K}^{0,2}_\mathsf{v})$ and P_2 holds $[\![\mathsf{v}]\!]^{\mathbf{G}}_2 = (\mathsf{K}^{0,1}_\mathsf{v}, \mathsf{K}^{\mathsf{v},2}_\mathsf{v})$. Here, $\mathsf{K}^{\mathsf{v},j}_\mathsf{v} = \mathsf{K}^{0,j}_\mathsf{v} \oplus \mathsf{v}\Delta^j$ for $j \in \{1, 2\}$, and Δ^j, which is known only to the garblers in Φ_j, denotes the global offset with its least significant bit set to 1 and is same for every wire in the circuit. A value $\mathsf{x} \in \mathbb{Z}_2$ is said to be $[\![\cdot]\!]^{\mathbf{C}}$-shared (compound shared) if each value from $(\mathsf{m_x}, \alpha_\mathsf{x}, \lambda^3_\mathsf{x})$, which are as defined above, is $[\![\cdot]\!]^{\mathbf{G}}$-shared. We write $[\![\mathsf{x}]\!]^{\mathbf{C}} = ([\![\mathsf{m_x}]\!]^{\mathbf{G}}, [\![\alpha_\mathsf{x}]\!]^{\mathbf{G}}, [\![\lambda^3_\mathsf{x}]\!]^{\mathbf{G}})$.

c) Generation of $[\![\mathsf{v}]\!]^{\mathbf{G}}$ and $[\![\mathsf{x}]\!]^{\mathbf{C}}$: Protocol $\Pi^{\mathbf{G}}_{\mathsf{Sh}}(\mathcal{P}, \mathsf{v})$ (Fig. 5) enables generation of $[\![\mathsf{v}]\!]^{\mathbf{G}}$ where two garblers in each garbling instance hold v, and proceeds as follows. Consider the first garbling instance with evaluator P_1 where garblers P_k, P_l hold v. Garblers in Φ_1 generate $\{\mathsf{K}^{\mathsf{b},1}_\mathsf{v}\}_{\mathsf{b} \in \{0,1\}}$ which denotes the key for value b on wire v, following the free-XOR technique [41], [42]. P_k, P_l jsnd $\mathsf{K}^{\mathsf{v},1}_\mathsf{v}$ to evaluator P_1. Similar steps carried out with respect to the second garbling instance, at the end of which, garblers in Φ_2 possess $\{\mathsf{K}^{\mathsf{b},2}_\mathsf{v}\}_{\mathsf{b} \in \{0,1\}}$ while the evaluator P_2 holds $\mathsf{K}^{\mathsf{v},2}_\mathsf{v}$. Following this, the shares $[\![\mathsf{v}]\!]^{\mathbf{G}}_s$ held by $P_s \in \mathcal{P}$ are defined as $[\![\mathsf{v}]\!]^{\mathbf{G}}_0 = [\![\mathsf{v}]\!]^{\mathbf{G}}_3 = (\mathsf{K}^{0,1}_\mathsf{v}, \mathsf{K}^{0,2}_\mathsf{v})$, $[\![\mathsf{v}]\!]^{\mathbf{G}}_1 = (\mathsf{K}^{\mathsf{v},1}_\mathsf{v}, \mathsf{K}^{0,2}_\mathsf{v})$, $[\![\mathsf{v}]\!]^{\mathbf{G}}_2 = (\mathsf{K}^{0,1}_\mathsf{v}, \mathsf{K}^{\mathsf{v},2}_\mathsf{v})$.

To generate $[\![\mathsf{x}]\!]^{\mathbf{C}}$, we need a way to generate $([\![\mathsf{m_x}]\!]^{\mathbf{G}}, [\![\alpha_\mathsf{x}]\!]^{\mathbf{G}}, [\![\lambda^3_\mathsf{x}]\!]^{\mathbf{G}})$, given $[\![\mathsf{x}]\!]^{\mathbf{B}}$. For this, $\Pi^{\mathbf{G}}_{\mathsf{Sh}}$ is invoked for each of $\mathsf{m_x}, \alpha_\mathsf{x}, \lambda^3_\mathsf{x}$.

Protocol $\Pi^{\mathbf{G}}_{\mathsf{Sh}}(\mathcal{P}, \mathsf{v})$

1) Garblers in Φ_j for $j \in \{1, 2\}$ generate keys $\mathsf{K}^{0,j}_\mathsf{v}, \mathsf{K}^{1,j}_\mathsf{v}$ for wire v, using free-XOR technique.
2) Let P^j_k, P^j_l denote the garblers in the j^{th} instance, for $j \in \{1, 2\}$, who hold $\mathsf{v} \in \mathbb{Z}_2$. P^j_k, P^j_l jsnd $\mathsf{K}^{\mathsf{v},j}_\mathsf{v}$ to evaluator P_j.
3) $P_i \in \{P_0, P_3\}$ sets $[\![\mathsf{v}]\!]^{\mathbf{G}}_i = (\mathsf{K}^{0,1}_\mathsf{v}, \mathsf{K}^{0,2}_\mathsf{v})$, P_1 sets $[\![\mathsf{v}]\!]^{\mathbf{G}}_1 = (\mathsf{K}^{\mathsf{v},1}_\mathsf{v}, \mathsf{K}^{0,2}_\mathsf{v})$ and P_2 sets $[\![\mathsf{v}]\!]^{\mathbf{G}}_2 = (\mathsf{K}^{0,1}_\mathsf{v}, \mathsf{K}^{\mathsf{v},2}_\mathsf{v})$.

Figure 5: Generation of $[\![\mathsf{v}]\!]^{\mathbf{G}}$

B. Conversions involving Garbled World

Assume the GC is required to compute a function f on inputs $\mathsf{x}, \mathsf{y} \in \mathbb{Z}_{2^\ell}$ and let the output be $f(\mathsf{x}, \mathsf{y})$. All the conversions described are for the 2 GC variant. Conversions for the 1 GC variant are straightforward, hence we omit the details. The conversions are generic for fair and robust variants, where the security follows from that of the underlying primitives.

Case I: Boolean-Garbled-Boolean. Since the inputs to the GC are available in boolean form, say $[\![\mathsf{x}]\!]^{\mathbf{B}}, [\![\mathsf{y}]\!]^{\mathbf{B}}$, parties generate $[\![\mathsf{x}]\!]^{\mathbf{C}}, [\![\mathsf{y}]\!]^{\mathbf{C}}$ by invoking the garbled sharing protocol $\Pi^{\mathbf{G}}_{\mathsf{Sh}}$. Additionally, parties P_0, P_3 sample $\mathsf{R} \in \mathbb{Z}_{2^\ell}$ to mask the function output, $f(\mathsf{x}, \mathsf{y})$, and generate $[\![\mathsf{R}]\!]^{\mathbf{B}}$ (using the joint sharing protocol) and $[\![\mathsf{R}]\!]^{\mathbf{G}}$. Garblers $P_g \in \{P_0, P_2, P_3\}$ garble the circuit which computes $\mathsf{z} = f(\mathsf{x}, \mathsf{y}) \oplus \mathsf{R}$, and send the GC along with the decoding information to evaluator P_1. Analogous steps are performed for evaluator P_2. Upon GC evaluation and output decoding, evaluators obtain $\mathsf{z} = f(\mathsf{x}, \mathsf{y}) \oplus \mathsf{R}$, and jointly boolean share z to generate $[\![\mathsf{z}]\!]^{\mathbf{B}}$. Parties then compute $[\![f(\mathsf{x}, \mathsf{y})]\!]^{\mathbf{B}} = [\![\mathsf{z}]\!]^{\mathbf{B}} \oplus [\![\mathsf{R}]\!]^{\mathbf{B}}$.

Case II: Boolean-Garbled-Arithmetic. This is similar to *Case I* except that the circuit which computes $\mathsf{z} = f(\mathsf{x}, \mathsf{y}) + \mathsf{R}$ is garbled instead. Boolean sharing of z is replaced with arithmetic, followed by computing $[\![f(\mathsf{x}, \mathsf{y})]\!] = [\![\mathsf{z}]\!] - [\![\mathsf{R}]\!]$.

Cases III & IV: Input in Arithmetic Sharing. The function to be computed $f(\mathsf{x}, \mathsf{y})$, is modified as

$f'(\mathsf{m_x}, \alpha_\mathsf{x}, \lambda^3_\mathsf{x}, \mathsf{m_y}, \alpha_\mathsf{y}, \lambda^3_\mathsf{y}) = f(\mathsf{m_x} - \alpha_\mathsf{x} - \lambda^3_\mathsf{x}, \mathsf{m_y} - \alpha_\mathsf{y} - \lambda^3_\mathsf{y})$ where inputs x, y are replaced by the triples $\{\mathsf{m_x}, \alpha_\mathsf{x}, \lambda^3_\mathsf{x}\}, \{\mathsf{m_y}, \alpha_\mathsf{y}, \lambda^3_\mathsf{y}\}$ and $\alpha_\mathsf{x} = \lambda^1_\mathsf{x} + \lambda^2_\mathsf{x}$ and $\alpha_\mathsf{y} = \lambda^1_\mathsf{y} + \lambda^2_\mathsf{y}$. The circuit to be garbled thus, corresponds to the function f'. Parties generate $[\![\mathsf{m_x}]\!]^\mathbf{G}, [\![\alpha_\mathsf{x}]\!]^\mathbf{G}, [\![\lambda^3_\mathsf{x}]\!]^\mathbf{G}, [\![\mathsf{m_y}]\!]^\mathbf{G}, [\![\alpha_\mathsf{y}]\!]^\mathbf{G}, [\![\lambda^3_\mathsf{y}]\!]^\mathbf{G}$ via $\Pi^\mathbf{G}_\mathsf{Sh}$, following which, parties proceed with the rest of the computation whose steps are similar to *Case I*, and *II*, depending on the requirement on the output sharing.

C. Other Conversions

a) Arithmetic to Boolean: To convert arithmetic sharing of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ to boolean sharing, observe that $\mathsf{v} = \mathsf{v}_1 + \mathsf{v}_2$ where $\mathsf{v}_1 = \mathsf{m_v} - \lambda^3_\mathsf{v}$ is possessed by parties P_1, P_2, while $\mathsf{v}_2 = -(\lambda^1_\mathsf{v} + \lambda^2_\mathsf{v})$ is possessed by parties P_0, P_3. Thus, $[\![\mathsf{v}]\!]^\mathbf{B}$ can be computed as $[\![\mathsf{v}]\!]^\mathbf{B} = [\![\mathsf{v}_1]\!]^\mathbf{B} + [\![\mathsf{v}_2]\!]^\mathbf{B}$, where $[\![\mathsf{v}_2]\!]^\mathbf{B}$ can be generated in the preprocessing phase, and $[\![\mathsf{v}_1]\!]^\mathbf{B}$ can be generated in the online phase by the respective parties executing joint boolean sharing protocol. The protocol appears in Fig. 20. Boolean addition, when instantiated using the adder of ABY2.0 [23], requires $\log_4(\ell)$ rounds.

b) Boolean to Arithmetic: To convert a boolean sharing of v into an arithmetic sharing, we use techniques from [4], [14]. For a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, note that

$$\mathsf{v} = \sum_{i=0}^{\ell-1} 2^i \mathsf{v}_i = \sum_{i=0}^{\ell-1} 2^i (\lambda_{\mathsf{v}_i} \oplus \mathsf{m}_{\mathsf{v}_i}) = \sum_{i=0}^{\ell-1} 2^i \left(\mathsf{m}_{\mathsf{v}_i}^\mathsf{R} + \lambda_{\mathsf{v}_i}^\mathsf{R}(1 - 2\mathsf{m}_{\mathsf{v}_i}^\mathsf{R})\right)$$

where $\lambda_{\mathsf{v}_i}^\mathsf{R}, \mathsf{m}_{\mathsf{v}_i}^\mathsf{R}$ denote the arithmetic value of bits $\lambda_{\mathsf{v}_i}, \mathsf{m}_{\mathsf{v}_i}$ over the ring $\mathbb{Z}_{2^\ell}$. For each bit v_i of v, parties generate the arithmetic sharing of $\lambda_{\mathsf{v}_i}^\mathsf{R}$ in the preprocessing, using techniques from bit to arithmetic protocol (cf. §V). During the online phase, additive shares for each bit v_i is locally computed similar to bit to arithmetic protocol. Parties then multiply the ith share with 2^i and locally add up to obtain an additive sharing of v. The rest of the steps are similar to the bit to arithmetic protocol, and the formal protocol appears in Fig. 21.

V. Building Blocks

This section covers the primitives needed for realising privacy-preserving variants of the applications considered, and elaborate details appear in §C. The building blocks can be combined to construct different layers in a neural network, as shown in [10] (Fig. 3).

a) Dot Product (Scalar Product): Given $[\![\vec{\mathsf{a}}]\!], [\![\vec{\mathsf{b}}]\!]$ with $|\vec{\mathsf{a}}| = |\vec{\mathsf{b}}| = \mathsf{d}$, protocol Π_dotp (Fig. 6) computes $[\![\mathsf{z}]\!]$ such that $\mathsf{z} = (\vec{\mathsf{a}} \odot \vec{\mathsf{b}})^\mathsf{t}$ if truncation is enabled, else $\mathsf{z} = \vec{\mathsf{a}} \odot \vec{\mathsf{b}}$. Following [4], [14], we combine the partial products from the multiplication protocol across d multiplications and communicate them in a single shot. This makes the communication cost of the dot product independent of the vector size. The protocol for robust setting follows similarly.

Matrix multiplication is an extension of the dot product protocol. We abuse notation and follow the $[\![\cdot]\!]$-sharing semantics (ref. §II) for matrices as well. For $\mathbf{X}^{u \times v}$, we have $\mathsf{m}_\mathbf{X} = \mathbf{X} \bigoplus [\lambda^1_\mathbf{X}] \bigoplus [\lambda^2_\mathbf{X}] \bigoplus [\lambda^3_\mathbf{X}]$. Here $\mathsf{m}_\mathbf{X}$, $[\lambda^1_\mathbf{X}]$, $[\lambda^2_\mathbf{X}]$, and $[\lambda^3_\mathbf{X}]$ are matrices of dimension $u \times v$, and $\bigoplus$ denote the matrix addition operation. Looking ahead $\ominus, \bigodot$ will be used to denote matrix subtraction and multiplication operation, respectively. Multiplication of two matrices, $\mathbf{X}^{u \times v}$, $\mathbf{Y}^{v \times w}$ is a collection of uw independent dot product operations over vectors of length v.

Protocol $\Pi_\mathsf{dotp}(\vec{\mathsf{a}}, \vec{\mathsf{b}}, \mathsf{isTr})$

Let isTr be a bit that denotes whether truncation is required ($\mathsf{isTr} = 1$) or not ($\mathsf{isTr} = 0$).

Preprocessing:

1) Locally compute:

$$P_0, P_1 : \gamma^1_{\vec{\mathsf{a}}\vec{\mathsf{b}}} = \sum_{i=1}^{\mathsf{d}} (\lambda^1_{\mathsf{a}_i}\lambda^3_{\mathsf{b}_i} + \lambda^3_{\mathsf{a}_i}\lambda^1_{\mathsf{b}_i} + \lambda^3_{\mathsf{a}_i}\lambda^3_{\mathsf{b}_i})$$

$$P_0, P_2 : \gamma^2_{\vec{\mathsf{a}}\vec{\mathsf{b}}} = \sum_{i=1}^{\mathsf{d}} (\lambda^2_{\mathsf{a}_i}\lambda^3_{\mathsf{b}_i} + \lambda^3_{\mathsf{a}_i}\lambda^2_{\mathsf{b}_i} + \lambda^2_{\mathsf{a}_i}\lambda^2_{\mathsf{b}_i})$$

$$P_0, P_3 : \gamma^3_{\vec{\mathsf{a}}\vec{\mathsf{b}}} = \sum_{i=1}^{\mathsf{d}} (\lambda^1_{\mathsf{a}_i}\lambda^2_{\mathsf{b}_i} + \lambda^2_{\mathsf{a}_i}\lambda^1_{\mathsf{b}_i} + \lambda^1_{\mathsf{a}_i}\lambda^1_{\mathsf{b}_i})$$

2) P_0, P_3 and P_j sample random $\mathsf{u}^j \in_R \mathbb{Z}_{2^\ell}$ for $j \in \{1, 2\}$. Let $\mathsf{u}^1 + \mathsf{u}^2 = \gamma^3_{\vec{\mathsf{a}}\vec{\mathsf{b}}} - \mathsf{r}$ for a random $\mathsf{r} \in_R \mathbb{Z}_{2^\ell}$.
3) P_0, P_3 compute $\mathsf{r} = \gamma^3_{\vec{\mathsf{a}}\vec{\mathsf{b}}} - \mathsf{u}^1 - \mathsf{u}^2$ and set $\mathsf{q} = \mathsf{r}^\mathsf{t}$ if $\mathsf{isTr} = 1$, else set $\mathsf{q} = \mathsf{r}$. P_0, P_3 execute $\Pi_\mathsf{JSh}(P_0, P_3, \mathsf{q})$ to generate $[\![\mathsf{q}]\!]$.
4) P_0, P_1, P_2 sample random $\mathsf{s}_1, \mathsf{s}_2 \in_R \mathbb{Z}_{2^\ell}$ and set $\mathsf{s} = \mathsf{s}_1 + \mathsf{s}_2$[a]. P_0 sends $\mathsf{w} = \gamma^1_{\vec{\mathsf{a}}\vec{\mathsf{b}}} + \gamma^2_{\vec{\mathsf{a}}\vec{\mathsf{b}}} + \mathsf{s}$ to P_3.

Online: Let $\mathsf{y} = (\mathsf{z} - \mathsf{r}) - \sum_{i=1}^{\mathsf{d}} \mathsf{m}_{\mathsf{a}_i}\mathsf{m}_{\mathsf{b}_i}$.

1) Locally compute:

$$P_1 : \mathsf{y}_1 = \sum_{i=1}^{\mathsf{d}} (-\lambda^1_{\mathsf{a}_i}\mathsf{m}_{\mathsf{b}_i} - \lambda^1_{\mathsf{b}_i}\mathsf{m}_{\mathsf{a}_i}) + \gamma^1_{\vec{\mathsf{a}}\vec{\mathsf{b}}} + \mathsf{u}^1$$

$$P_2 : \mathsf{y}_2 = \sum_{i=1}^{\mathsf{d}} (-\lambda^2_{\mathsf{a}_i}\mathsf{m}_{\mathsf{b}_i} - \lambda^2_{\mathsf{b}_i}\mathsf{m}_{\mathsf{a}_i}) + \gamma^2_{\vec{\mathsf{a}}\vec{\mathsf{b}}} + \mathsf{u}^2$$

$$P_1, P_2 : \mathsf{y}_3 = \sum_{i=1}^{\mathsf{d}} (-\lambda^3_{\mathsf{a}_i}\mathsf{m}_{\mathsf{b}_i} - \lambda^3_{\mathsf{b}_i}\mathsf{m}_{\mathsf{a}_i})$$

2) P_1 sends y_1 to P_2, while P_2 sends y_2 to P_1, and they locally compute $\mathsf{z} - \mathsf{r} = (\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{y}_3) + \sum_{i=1}^{\mathsf{d}} \mathsf{m}_{\mathsf{a}_i}\mathsf{m}_{\mathsf{b}_i}$.
3) If $\mathsf{isTr} = 1$, P_1, P_2 set $\mathsf{p} = (\mathsf{z} - \mathsf{r})^\mathsf{t}$, else $\mathsf{p} = \mathsf{z} - \mathsf{r}$. P_1, P_2 execute $\Pi_\mathsf{JSh}(P_1, P_2, \mathsf{p})$ to generate $[\![\mathsf{p}]\!]$.
4) Parties locally compute $[\![\mathsf{o}]\!] = [\![\mathsf{p}]\!] + [\![\mathsf{q}]\!]$. Here $\mathsf{o} = \mathsf{z}^\mathsf{t}$ if $\mathsf{isTr} = 1$ and z otherwise.
5) *Verification:* P_3 computes $\mathsf{v} = \sum_{i=1}^{\mathsf{d}}(-(\lambda^1_{\mathsf{a}_i} + \lambda^2_{\mathsf{a}_i})\mathsf{m}_{\mathsf{b}_i} - (\lambda^1_{\mathsf{b}_i} + \lambda^2_{\mathsf{b}_i})\mathsf{m}_{\mathsf{a}_i}) + \mathsf{u}^1 + \mathsf{u}^2 + \mathsf{w}$ and sends $\mathsf{H}(\mathsf{v})$ to P_1 and P_2. Parties P_1, P_2 `abort` iff $\mathsf{H}(\mathsf{v}) \neq \mathsf{H}(\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{s})$.

[a] For the fair protocol, it is enough for P_0, P_1, P_2 to sample s directly.

Figure 6: Dot Product with / without Truncation.

In a convolutional neural network, a convolution operation can be reduced to matrix multiplications [14], [46] as follows. Consider an $f \times f$ kernel over a $w \times h$ input with $p \times p$ padding using $s \times s$ stride having i input channels and o output channels. A convolution can be computed as a matrix multiplication on matrices of dimension $(w' \cdot h') \times (i \cdot f \cdot f)$ and $(i \cdot f \cdot f) \times (o)$ where $w' = \frac{w - f + 2p}{s} + 1$ and $h' = \frac{h - f + 2p}{s} + 1$.

b) Multi-input Multiplication: Inspired from ABY2.0 [23], we design 3-input and 4-input multiplication protocols for our setting. We remark that the multi-input multiplication, when coupled with the optimized PPA circuit from [23], improves the rounds as well as communication in the online phase.

The goal of 3-input multiplication is to generate $[\![\cdot]\!]$-sharing of $\mathsf{z} = \mathsf{abc}$ given $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!], [\![\mathsf{c}]\!]$, without the need for performing two sequential multiplications (i.e. first ab then abc). For this parties proceed similar to the multiplication protocol (see §III-B), where they compute $[\![\mathsf{z}]\!] = [\![\mathsf{z} - \mathsf{r}]\!] + [\![\mathsf{r}]\!]$. Observe that

$$\begin{aligned} \mathsf{z} - \mathsf{r} &= \mathsf{abc} - \mathsf{r} = (\mathsf{m_a} - \lambda_\mathsf{a})(\mathsf{m_b} - \lambda_\mathsf{b})(\mathsf{m_c} - \lambda_\mathsf{c}) - \mathsf{r} \\ &= \mathsf{m_{abc}} - \mathsf{m_{ac}}\lambda_\mathsf{b} - \mathsf{m_{bc}}\lambda_\mathsf{a} - \mathsf{m_{ab}}\lambda_\mathsf{c} + \mathsf{m_a}\gamma_\mathsf{bc} + \mathsf{m_b}\gamma_\mathsf{ac} \\ &\quad + \mathsf{m_c}\gamma_\mathsf{ab} - \gamma_\mathsf{abc} - \mathsf{r} \end{aligned}$$

Similar to the 2-input fair multiplication Π_{Mult} (Fig. 3), the goal of the preprocessing phase is to generate additive shares of $\gamma_\mathsf{ab}, \gamma_\mathsf{ac}, \gamma_\mathsf{bc}, \gamma_\mathsf{abc}$ among P_1, P_2.

Informally, the terms that P_1, P_2 cannot compute locally for the aforementioned γ values, can be computed by P_0, P_3, as evident from our sharing semantics. P_0, P_3 compute the missing terms and share them among P_1, P_2 in the preprocessing phase. P_1, P_2 proceed with online phase similar to Π_{Mult}, to compute $\mathsf{z} - \mathsf{r}$. Thus the online complexity is retained as that of Π_{Mult} while the preprocessing communication is increased to 9 elements. The protocol appears in Fig. 15.

For the 4-input case, the goal is to compute $\mathsf{z} = \mathsf{abcd}$ for which the additive shares of γ_ab, γ_ac, γ_ad, γ_bc, γ_bd, γ_cd, γ_abc, γ_acd, γ_bcd, γ_abcd needs to be generated in the preprocessing. The protocol is very similar to the 3-input case, and the details are deferred to §C.

c) Secure Comparison: To compute $\mathsf{a} > \mathsf{b}$ in the FPA representation, given its $[\![\cdot]\!]$-sharing, Π_{bitext} uses the technique of extracting the most significant bit (msb) of the value $\mathsf{v} = \mathsf{a} - \mathsf{b}$ [6], [8], [14]. To compute the msb, we use two variants - i) the communication optimized parallel prefix adder (PPA) circuit from ABY3 [6] ($2(\ell - 1)$ AND gates, $\log \ell$ depth), and ii) the round optimized bit extraction circuit from ABY2 [23]. The circuit of ABY2 uses multi-input AND gates and has a multiplicative depth of $\log_4(\ell)$. Both these circuits take two ℓ-bit values in boolean sharing as the input and outputs the result in boolean sharing form. Note that $\mathsf{v} = (\mathsf{m_v} - \lambda_\mathsf{v}^3) + (-\lambda_\mathsf{v}^1 - \lambda_\mathsf{v}^2)$ as per the sharing semantics (cf. Table III). P_0, P_3 execute $\Pi_{\mathsf{JSh}}^{\mathbf{B}}$ on $(-\lambda_\mathsf{v}^1 - \lambda_\mathsf{v}^2)$ during the preprocessing, while P_0, P_3 execute $\Pi_{\mathsf{JSh}}^{\mathbf{B}}$ on $(\mathsf{m_v} - \lambda_\mathsf{v}^3)$ during the online phase to generate the respective boolean sharing.

d) Bit to Arithmetic: Protocol Π_{bit2A} (Fig. 16) enables computing $[\![\mathsf{b}]\!]$ of a bit b given its boolean sharing $[\![\mathsf{b}]\!]^{\mathbf{B}}$. Let b^R denotes the value of $\mathsf{b} \in \{0, 1\}$ over the arithmetic ring $\mathbb{Z}_{2^\ell}$. Then for $\mathsf{b} = \mathsf{b}_1 \oplus \mathsf{b}_2$, note that $\mathsf{b}^\mathsf{R} = (\mathsf{b}_1^\mathsf{R} - \mathsf{b}_2^\mathsf{R})^2$.

Let $\mathsf{b}_1 = \mathsf{m_b} \oplus \lambda_\mathsf{v}^3$ and $\mathsf{b}_2 = \lambda_\mathsf{v}^1 \oplus \lambda_\mathsf{v}^2$. To compute $[\![\mathsf{b}]\!]$, a pair of parties can generate the arithmetic sharing corresponding to b_1^R and b_2^R by executing Π_{JSh}. $[\![\mathsf{b}]\!]$ can be computed by invoking Π_{Mult} once with inputs $\mathsf{x} = \mathsf{y} = \mathsf{b}_1^\mathsf{R} - \mathsf{b}_2^\mathsf{R}$.

Using the techniques from [4], [14], we obtain a communication-optimized variant by trading off computation in the preprocessing. For this, note that

$$\mathsf{b}^\mathsf{R} = (\mathsf{m_b} \oplus \lambda_\mathsf{b})^\mathsf{R} = \mathsf{m_b^R} + (\lambda_\mathsf{b})^\mathsf{R}(1 - 2\mathsf{m_b^R}) \quad (3)$$

Let $\mathsf{v} = \mathsf{m_b^R}$ and $\mathsf{u} = (\lambda_\mathsf{b})^\mathsf{R}$. During the preprocessing, P_0 generates $\langle\cdot\rangle$-sharing of u and a check is executed to verify its correctness. The online phase consists of each pair of parties (P_1, P_3), (P_2, P_3) and (P_1, P_2) locally computing an additive sharing of b^R, generating the corresponding $[\![\cdot]\!]$-sharing using Π_{JSh}, and locally adding the shares to obtain $[\![\mathsf{b}]\!]$.

e) Bit Injection: Protocol Π_{bitInj} enables computing $[\![\mathsf{bv}]\!]$, given the boolean sharing $[\![\mathsf{b}]\!]^{\mathbf{B}}$ of a bit b and the arithmetic sharing $[\![\mathsf{v}]\!]$ of a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$. Similar to Π_{bit2A},

$$\begin{aligned} (\mathsf{bv})^\mathsf{R} &= (\mathsf{m_b} \oplus \lambda_\mathsf{b})^\mathsf{R}(\mathsf{m_v} - \lambda_\mathsf{v}) \\ &= (\mathsf{m_b^R} + (\lambda_\mathsf{b})^\mathsf{R}(1 - 2\mathsf{m_b^R}))(\mathsf{m_v} - \lambda_\mathsf{v}) \\ &= \mathsf{m_b^R}\mathsf{m_v} - \mathsf{m_b^R}\lambda_\mathsf{v} + (2\mathsf{m_b^R} - 1)((\lambda_\mathsf{b})^\mathsf{R}\lambda_\mathsf{v} - \mathsf{m_v}(\lambda_\mathsf{b})^\mathsf{R}) \end{aligned}$$

During preprocessing, P_0 generates $\langle\cdot\rangle$-sharing of $\lambda_\mathsf{b}^\mathsf{R}$, followed by verifying its correctness, similar to Π_{bit2A}. $\langle\cdot\rangle$-shares of $(\lambda_\mathsf{b})^\mathsf{R}\lambda_\mathsf{v}$ are generated by multiplying $\langle(\lambda_\mathsf{b})^\mathsf{R}\rangle$ and $\langle\lambda_\mathsf{v}\rangle$ using Π_{MulR} (Fig. 2). In the online phase, each pair of parties (P_1, P_3), (P_2, P_3) and (P_1, P_2) locally compute an additive sharing of $(\mathsf{bv})^\mathsf{R}$, generate its $[\![\cdot]\!]$-sharing using Π_{JSh}, and locally add these shares to generate $[\![(\mathsf{bv})^\mathsf{R}]\!]$.

f) Oblivious Selection: Given $[\![\cdot]\!]$-shares of $\mathsf{x}_0, \mathsf{x}_1 \in \mathbb{Z}_{2^\ell}$ and $[\![\mathsf{b}]\!]^{\mathbf{B}}$ where $\mathsf{b} \in \{0, 1\}$, oblivious selection (Π_{obv}) enables parties to generate re-randomized $[\![\cdot]\!]$-shares of $\mathsf{z} = \mathsf{x_b}$. The protocol is similar in spirit to Oblivious Transfer primitive. Note that z can be written as $\mathsf{z} = \mathsf{b}(\mathsf{x}_1 - \mathsf{x}_0) + \mathsf{x}_0$. Parties invoke Π_{bitInj} to compute $[\![\mathsf{b}(\mathsf{x}_1 - \mathsf{x}_0)]\!]$, and sum it with $[\![\mathsf{x}_0]\!]$ to generate $[\![\mathsf{z}]\!]$.

g) Piece-wise Polynomials: Piece-wise polynomial functions are constructed as a series of constant public polynomials $f_1, \ldots, f_m$ and $c_1 < \ldots < c_m$ such that,

$$f(y) = \begin{cases} 0, & y < c_1 \\ f_1, & c_1 \le y < c_2 \\ \ldots \\ f_m, & c_m \le y \end{cases}$$

f can be computed as, $f(y) = \sum_{i=1}^{m} \mathsf{b}_i \cdot (f_i - f_{i-1})$, where $f_0 = 0$, $f_m = 1$, and $\mathsf{b}_i = 1$ if $y \ge c_i$ and 0 otherwise, for $i \in \{1, \ldots, m\}$. Given the $[\![\cdot]\!]$-shares of y, one can obtain the $[\![\cdot]\!]^{\mathbf{B}}$-shares of the bits $\mathsf{b}_1, \ldots, \mathsf{b}_m$ using secure comparison. Shares of the product terms, $\mathsf{b}_i \cdot (f_i - f_{i-1})$, can thus be generated by invoking m Π_{bitInj}, followed by a local addition. A naive application of Π_{bitInj} involves sharing (via Π_{JSh}) additive shares of $\mathsf{b}_i \cdot (f_i - f_{i-1})$, thereby requiring m

Π_{JSh} in the online phase. Instead, it can be made independent of m by first computing additive shares of $f(y)$, and then invoking one Π_{JSh}.

Non-linear activation functions, such as Rectified Linear Unit and Sigmoid, can be viewed as instantiations of piece-wise polynomial functions as shown in ABY3 [6].

h) ArgMin/ ArgMax: Protocol Π_{argmin} (Fig. 18) allows parties to compute the index of the smallest element in a vector $\vec{\mathsf{x}} = (\mathsf{x}_1, \ldots, \mathsf{x}_m)$ of m elements, where $\vec{\mathsf{x}}$ is $[\![\cdot]\!]$-shared, i.e. each element $\mathsf{x}_i \in \mathbb{Z}_{2^\ell}$ of $\vec{\mathsf{x}}$ is $[\![\cdot]\!]$-shared. The protocol outputs a $[\![\cdot]\!]^{\mathbf{B}}$-shared bit vector $\vec{\mathsf{b}}$ of size m which has a 1 at the index associated with the minimum value in $\vec{\mathsf{x}}$, and 0 elsewhere. We follow the standard tree-based approach [18] to recursively find the minimum value in $\vec{\mathsf{x}}$ while also updating $\vec{\mathsf{b}}$ to reflect the index of this smallest element. Each bit of $\vec{\mathsf{b}}$ is initialized to 1. The elements of $\vec{\mathsf{x}}$ are grouped into pairs and securely compared to find their pairwise minimum. Using this information, $\vec{\mathsf{b}}$ is updated such that b_j's are reset to 0 for x_j's $\in \vec{\mathsf{x}}$ which do not form the minimum in their respective pair; the other bits in $\vec{\mathsf{b}}$ still equal 1. The protocol recurses on the remaining elements $\mathsf{x}_j \in \vec{\mathsf{x}}$, which were the pairwise minimums. Eventually, only one $\mathsf{b}_j \in \vec{\mathsf{b}}$ equals 1, indicating that x_j is the minimum, with index j. Computing Π_{argmax} can be done similarly.

VI. Implementation and Benchmarking

We benchmark training and inference phases for deep NNs with varying parameter sizes and the inference phase for Support Vector Machines (SVM) using MNIST [47] and CIFAR-10 [48] dataset. Training phase of SVM requires additional tools and primitives, and is out of scope of this work. Benchmarks of the protocols are against the state-of-the-art 4PC of Trident [4] and SWIFT [14] 4PC (supports only inference).

a) Benchmarking Environment Details: The protocols are benchmarked over a Wide Area Network (WAN), instantiated using n1-standard-64 instances of Google Cloud[2], with machines located in East Australia (P_0), South Asia (P_1), South East Asia (P_2), and West Europe (P_3). The machines are equipped with 2.0 GHz Intel (R) Xeon (R) (Skylake) processors supporting hyper-threading, with 64 vCPUs, and 240 GB of RAM Memory. Parties are connected by pair-wise authenticated bidirectional synchronous channels (e.g., instantiated via TLS over TCP/IP). We use a bandwidth of 40 MBps between every pair of parties and the average round-trip time (rtt)[3] values among P_0-P_1, P_0-P_2, P_0-P_3, P_1-P_2, P_1-P_3, and P_2-P_3 are $153.74ms$, $93.39ms$, $274.84ms$, $62.01ms$, $174.15ms$, and $219.46ms$ respectively.

For a fair comparison, we implemented and benchmarked all the protocols, including the protocols of Trident and SWIFT, building on the ENCRYPTO library [49] in C++17. Primitives such as maxpool, which Trident and SWIFT do not support, have been run using our building blocks. We would like to clarify that our code is developed for benchmarking, is not optimized for industry-grade use, and optimizations like GPU support can further enhance performance. Our protocols are instantiated over a 64-bit ring ($\mathbb{Z}_{2^{64}}$), and the collision-resistant hash function is instantiated using SHA-256. We use multi-threading, and our machines are capable of handling a total of 64 threads. Each experiment is run 10 times, and the average values are reported. We use 1 KB = 8192 bits and use a batch size of $B = 128$ for training.

b) Benchmarking Parameters: We evaluate the protocols across a variety of parameters as given in Table V. In addition to parameters such as runtime, communication, and *online throughput* (TP) [4], [6], [19], [21], the cumulative runtime (sum of the up-time of all the hired servers) is also reported. This is because when deployed over third-party cloud servers, one pays for them by the communication and the uptime of the hired servers. To analyze the cost of deployment of the framework, *monetary cost* (Cost) [50] is reported. This is done using the pricing of Google Cloud Platform[4], where for 1 GB and 1 hour of usage, the costs are USD 0.08 and USD 3.04, respectively. For protocols with an asymmetric communication graph, communication load is unevenly distributed among all the servers, leaving several communication channels underutilized. Load balancing improves the performance by running several execution threads in parallel, each with the roles of the servers changed. Load balancing has been performed in all the protocols benchmarked.

Notation	Description
$\mathsf{T}_{\mathsf{on,i}}$	Online runtime of party P_i.
$\mathsf{T}_{\mathsf{tot,i}}$	Total runtime of party P_i.
$\mathsf{PT}_{\mathsf{on}}$	Protocol online runtime; $\mathsf{max_i}\{\mathsf{T}_{\mathsf{on,i}}\}$.
$\mathsf{PT}_{\mathsf{tot}}$	Protocol total runtime; $\mathsf{max_i}\{\mathsf{T}_{\mathsf{tot,i}}\}$.
$\mathsf{CT}_{\mathsf{on}}$	Cumulative online runtime; $\Sigma_i \mathsf{T}_{\mathsf{on,i}}$.
$\mathsf{CT}_{\mathsf{tot}}$	Cumulative total runtime; $\Sigma_i \mathsf{T}_{\mathsf{tot,i}}$.
$\mathsf{Comm}_{\mathsf{on}}$	Online communication.
$\mathsf{Comm}_{\mathsf{tot}}$	Total communication.
Cost	Total monetary cost.
TP	Online throughput (higher = better) (#iterations / #queries per minute in online)

Table V: Benchmarking parameters (lower is better, except for TP)

c) Network Architectures: We consider the following networks for benchmarking. These were chosen based on the different range of model parameters and types of layers used in the networks. We refer readers to [7], [51] for the architecture and a detailed description of the training and inference steps for the ML algorithms.

– *SVM:* Consists of 10 categories for classification [18].

– *NN-1:* Fully connected network with 3 layers and around 118K parameters [6], [8].

– *NN-2:* Convolutional neural network comprising of 2 hidden layers, with 100 and 10 nodes [4], [6], [10].

– *NN-3:* LeNet [32], comprises of 2 convolutional and fully connected layers, followed by maxpool for convolutional layers. This has approximately 431K parameters.

– *NN-4:* VGG16 [33] has 16 layers in total and contains fully-connected, convolutional, ReLU activation and maxpool layers. This has ≈138 million parameters.

[2] https://cloud.google.com/

[3] Time for communicating 1 KB of data between a pair of parties

[4] See https://cloud.google.com/vpc/network-pricing for network cost and https://cloud.google.com/compute/vm-instance-pricing for computation cost.

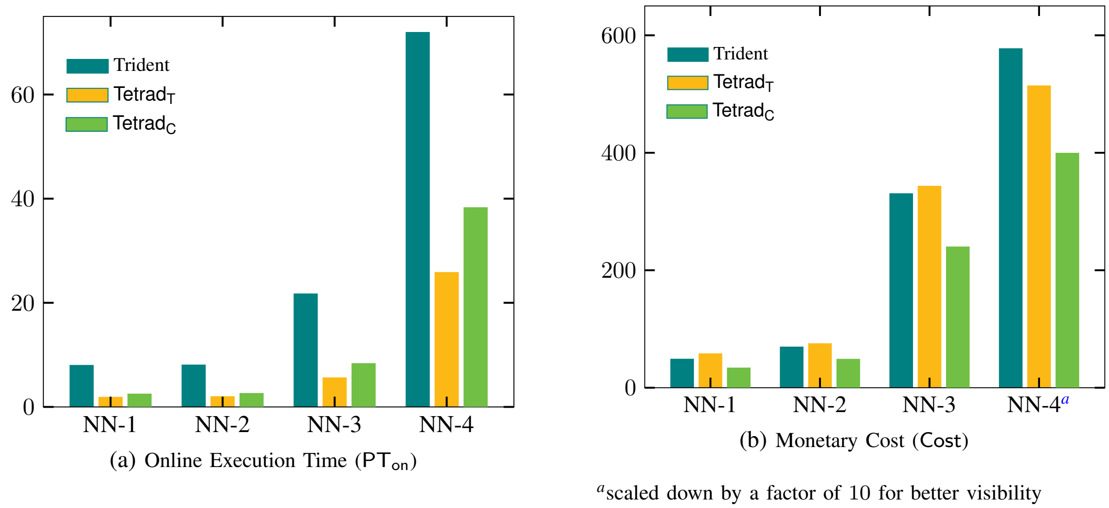

Figure 7: Training of Neural Networks: in terms of $\mathsf{PT_{on}}$ and Cost (lower is better) (cf. Table V)

d) Datasets: We use the following datasets:

– MNIST [47] is a collection of 28×28 pixel, handwritten digit images with a label between 0 and 9 for each. It has 60,000 and respectively, 10,000 images in training and test set. We evaluate NN-1, NN-3, SVM on this dataset.

– CIFAR-10 [48] has 32×32 pixel images of 10 different classes such as dogs, horses, etc. It has 50,000 images for training and 10,000 for testing, with 6,000 images in each class. NN-2, NN-4 are evaluated on this dataset.

e) Discussion: Broadly speaking, we consider two deployment scenarios – optimized for time (T), and for cost (C). In the first one, participants want the result of the output as soon as possible while maximizing the online throughput. In the second one, they want the overall monetary cost of the system to be minimal and are willing to tolerate an overhead in the execution time. Using multi-input multiplication gates and the 2 GC variant of the garbled makes the online phase faster but incur an increase in monetary cost. This is because they cause an overhead in communication in the preprocessing phase, and communication affects monetary cost more than uptime (in our setting).

$\mathsf{Tetrad_T}$ makes use of multi-input multiplication gates and the 2 GC variant of the garbled world and is the fastest variants of the framework. On the other hand, $\mathsf{Tetrad_C}$ is the variant with minimal monetary cost. We only report the numbers for the fair variant of Tetrad and not the robust variant. The overhead for the robust variant over the fair one is minimal, and is primarily due to (i) the use of *robust* joint-send primitive and (ii) the augmented one-time verification check at the end of the preprocessing phase. The overhead amortises for deep networks, like the ones considered in this work.

A. ML Training

For training we consider NN-1, NN-2, NN-3 and NN-4 networks. We report values corresponding to one iteration, that comprises of a forward propagation followed by a backward propagation. More details are provided in §F.

Starting with the time-optimized variant, $\mathsf{Tetrad_T}$ is $3-4\times$ faster than Trident in online runtime. The primary factor is the reduction in online rounds of our protocol due to multi-input gates. More precisely, we use the depth-optimized bit extraction circuit while instantiating the ReLU activation function using multi-input AND gates (cf. §V). Looking at the total communication ($\mathsf{Comm_{tot}}$) in Table VI, we observe that the gap in $\mathsf{Comm_{tot}}$ between $\mathsf{Tetrad_T}$ vs. Trident decreases as the networks get deeper. This is justified as the improvement in communication of our dot product with truncation outpaces the overhead in communication caused by multi-input gates. The impact of this is more pronounced with NN-4, as observed by the lower monetary cost of $\mathsf{Tetrad_T}$ over Trident. Another reason is that there are two active parties

Algorithm	Parameter	Trident	$\mathsf{Tetrad_T}$	$\mathsf{Tetrad_C}$
NN-1	$\mathsf{PT_{on}}$	8.06	1.93	2.55
	$\mathsf{PT_{tot}}$	10.76	5.05	5.27
	$\mathsf{CT_{tot}}$	27.90	12.69	11.22.
	$\mathsf{Comm_{tot}}$	0.16	0.30	0.16
	Cost	49.33	58.51	34.29
	TP	1904.79	3792.64	3725.49
NN-2	$\mathsf{PT_{on}}$	8.13	2.05	2.67
	$\mathsf{PT_{tot}}$	11.47	5.79	6.14
	$\mathsf{CT_{tot}}$	30.88	14.82	13.40
	$\mathsf{Comm_{tot}}$	0.28	0.39	0.24
	Cost	70.00	75.67	49.16
	TP	428.16	652.75	644.69
NN-3	$\mathsf{PT_{on}}$	21.79	5.67	8.40
	$\mathsf{PT_{tot}}$	30.66	15.14	17.87
	$\mathsf{CT_{tot}}$	91.68	40.01	42.76
	$\mathsf{Comm_{tot}}$	1.59	1.94	1.28
	Cost	331.01	343.73	240.41
	TP	53.62	55.71	54.13
NN-4	$\mathsf{PT_{on}}$	72.01	25.90	38.35
	$\mathsf{PT_{tot}}$	283.89	182.13	194.58
	$\mathsf{CT_{tot}}$	859.09	500.13	522.32
	$\mathsf{Comm_{tot}}$	31.59	29.52	22.24
	Cost	5779.27	5146.10	3999.30
	TP	2.55	2.61	2.56

Table VI: Benchmarking of the training phase of ML algorithms. Time (in seconds) and communication (in GB) are reported for 1 iteration. Monetary cost (USD) is reported for 1000 iterations.

(P_1, P_2) in our framework, whereas Trident has three. Given the allocation of servers, the best rtt Trident can get with three parties (P_0, P_1, P_2) is $153.74ms$, as compared to $62.01ms$ of Tetrad, contributing to Tetrad being faster. However, if the rtt among all the parties were similar, this gap would be closed. Concretely, the online runtime ($\mathsf{PT_{on}}$) of Trident will be similar to that of $\mathsf{Tetrad_C}$.

The cost-optimized variant $\mathsf{Tetrad_C}$ on the other hand, is $1.5\times$ slower in the online phase compared to $\mathsf{Tetrad_T}$. However, it is still faster than Trident owing to the rtt setup, as discussed above. When it comes to monetary cost, this variant is up to $20-40\%$ cheaper than it's time-optimized counterpart and cheaper by around 30% over Trident.

These trends can be better captured with a pictorial representation as given in Figure 7.

a) Varying batch sizes and feature sizes: Table VII shows the online throughput (TP) of neural network (NN-1) training over varying batch sizes and feature sizes using synthetic datasets.

Batch Size	Features	Trident	$\mathsf{Tetrad_T}$	$\mathsf{Tetrad_C}$
128	10	1905.58	5407.35	5271.88
	100	1905.58	5152.29	5029.14
	1000	1904.4	3500.89	3443.6
256	10	1905.58	2818.4	2744.87
	100	1905.58	2747.5	2677.58
	1000	1849.78	2195.3	2150.43

Table VII: Online throughput (TP) of NN-1 training (iterations per minute) over various batch sizes and features.

We find that both $\mathsf{Tetrad_T}$, $\mathsf{Tetrad_C}$ are up to $1.8\times$ higher in TP. However, as the batch size and feature size increase, both Trident and Tetrad experience a bandwidth bottleneck. The effect of the bandwidth limitation is higher for Tetrad; hence the gain in TP over Trident decreases a bit.

B. ML Inference

We benchmark the inference phase of SVM and the aforementioned NNs. In addition to Trident [4], we also benchmark against the 4PC robust protocol of SWIFT [14] since it supports NN inference. Note that the best case performance of Fantastic Four [29] when cast in the preprocessing model resembles that of SWIFT, while their worst case execution (3PC malicious) is an order of magnitude slower (cf. §A-D), as demonstrated in their paper (cf. Table 2 of [29]).

Similar to training, the time-optimized variant for inference is faster when it comes to $\mathsf{PT_{on}}$, by $4-6\times$ over Trident. This is also reflected in the TP, where the improvement is about $2.8-5.5\times$, as evident from Figure 8. In inference, the communication is in the order of megabytes, while run time is in the order of a few seconds. The key observation is that communication is well suited for the bandwidth used (40 MBps). So unlike training, the monetary cost in inference depends more on run time rather than on communication. This is evident from Table VIII which shows that $\mathsf{Tetrad_T}$ saves on monetary cost up to a factor of 6 over Trident.

Note that the cost-optimized variant under performs in terms of monetary cost compared to $\mathsf{Tetrad_T}$. This is because,

Algorithm	Parameter	Trident	$\mathsf{Tetrad_T}$	$\mathsf{Tetrad_C}$	SWIFT
SVM	$\mathsf{PT_{on}}$	17.09	2.91	4.77	5.21
	$\mathsf{PT_{tot}}$	17.37	3.19	5.05	6.04
	$\mathsf{CT_{tot}}$	47.02	6.99	10.70	14.47
	$\mathsf{Comm_{tot}}$	1.36	2.34	1.25	1.36
	Cost	39.92	6.26	9.23	12.43
	TP	898.80	5271.74	3221.29	2949.76
NN-1	$\mathsf{PT_{on}}$	5.87	1.31	1.87	2.31
	$\mathsf{PT_{tot}}$	6.15	1.58	2.14	3.13
	$\mathsf{CT_{tot}}$	16.75	3.76	4.88	8.65
	$\mathsf{Comm_{tot}}$	0.06	0.09	0.05	0.06
	Cost	14.15	3.19	4.13	7.32
	TP	2615.35	11734.60	8226.93	6661.00
NN-2	$\mathsf{PT_{on}}$	5.87	1.31	1.87	2.31
	$\mathsf{PT_{tot}}$	6.15	1.58	2.14	3.13
	$\mathsf{CT_{tot}}$	16.75	3.77	4.88	8.66
	$\mathsf{Comm_{tot}}$	0.26	0.37	0.22	0.25
	Cost	14.19	3.24	4.16	7.35
	TP	2615.35	11734.60	8226.93	6661.00
NN-3	$\mathsf{PT_{on}}$	14.42	2.61	4.10	4.54
	$\mathsf{PT_{tot}}$	14.71	2.91	4.39	5.39
	$\mathsf{CT_{tot}}$	39.92	6.43	9.40	13.18
	$\mathsf{Comm_{tot}}$	5.62	8.42	4.76	5.39
	Cost	34.59	6.74	8.68	11.97
	TP	1065.35	5882.44	3746.89	3384.51
NN-4	$\mathsf{PT_{on}}$	47.05	7.85	12.69	13.13
	$\mathsf{PT_{tot}}$	47.61	8.44	13.28	14.33
	$\mathsf{CT_{tot}}$	129.41	17.77	27.46	31.35
	$\mathsf{Comm_{tot}}$	85.69	124.09	71.27	81.33
	Cost	122.66	34.40	34.32	39.18
	TP	326.46	934.34	891.19	891.19

Table VIII: Benchmarking of the inference phase of ML algorithms. Time (in seconds) and communication (in MB) are reported for 1 query. Monetary cost (USD) is reported for 1000 queries.

as mentioned earlier, run time plays a bigger role in monetary cost than communication. Hence for inference, the time-optimized variant becomes the optimal choice.

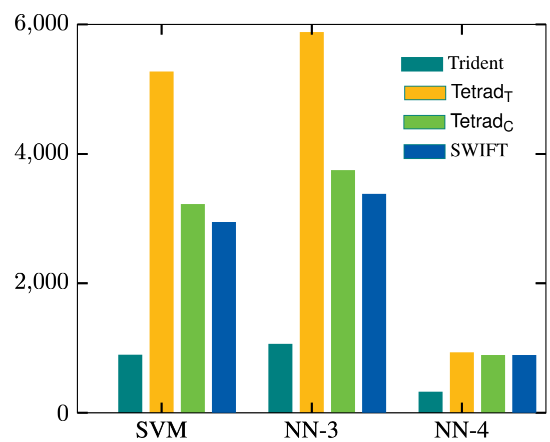

Figure 8: Inference of SVM, NN-3 and NN-4: in terms of TP (higher is better)

C. Comparison operations

Table IX compares the performance of the frameworks for circuits of varying depth. At each layer of the circuits, we perform 128 comparisons where the comparison results are generated in arithmetic shared form. The idea is that each layer emulates a comparison layer in an NN with a batch size of 128.

Depth	Parameter	Trident	$\mathsf{Tetrad}_\mathsf{T}$	$\mathsf{Tetrad}_\mathsf{C}$
128	PT_on	3.55	0.53	0.93
	CT_tot	9.6	1.06	1.85
	Cost	0.49	0.05	0.09
1024	PT_on	28.42	4.23	7.41
	CT_tot	76.79	8.47	14.82
	Cost	3.89	0.43	0.75
8192	PT_on	227.34	33.87	59.27
	CT_tot	614.3	67.76	118.56
	Cost	31.27	3.48	6.03

Table IX: Benchmarking of comparisons over various depths. Each of the layer has 128 comparisons. Time is reported in minutes, and monetary cost in USD.

Interestingly, beyond a depth of roughly 100, the time-optimized variant ($\mathsf{Tetrad}_\mathsf{T}$) starts outperforming in every metric, especially monetary cost, over the cost-optimized one ($\mathsf{Tetrad}_\mathsf{C}$). This is because as the depth increases, runtime (CT) grows at a much higher rate than the total communication. What we can infer from Table IX is that if one were to use a DNN with a depth of over 100, $\mathsf{Tetrad}_\mathsf{T}$ becomes the optimal choice.

Future Work

Tetrad requires the preprocessing to be function-dependent. Decoupling the preprocessing from the function to be computed in the online phase will make the framework more generic and is left as an interesting direction to pursue. Even though fixed-point arithmetic is efficient for the applications considered, in some cases, other representations such as floating-point and posit arithmetic might be desirable. Supporting alternative representations may require rethinking parts of the framework; hence it is left as an open problem.

The following are some of the challenges to be addressed while extending Tetrad to support training of other ML algorithms such as SVM, ResNet and LSTMs. In SVM training, the choice of kernel function plays an important role in determining the efficiency, especially for the non-linear classifiers. Some of the most widely used non-linear kernels include i) Polynomial: $(\vec{\mathbf{x}} \odot \vec{\mathbf{y}})^d$, ii) Gaussian: $\exp(-\gamma\|\vec{\mathbf{x}} - \vec{\mathbf{y}}\|^2)$ for $\gamma > 0$, and iii) Hyperbolic: $\tanh(\mu\vec{\mathbf{x}} \odot \vec{\mathbf{y}} + c)$ for some $\mu > 0$ and $c < 0$, where $\vec{\mathbf{x}}, \vec{\mathbf{y}}$ denote the input vectors. These kernels are expensive to compute (computation and communication) using standard MPC approaches such as circuit garbling, and hence, demand new MPC-friendly protocols which guarantee efficiency without losing out on accuracy (e.g., Sigmoid approximation of [7]). Further, note that using the naive MPC protocols for training would demand a non-linear increase in bit-size of fixed-point arithmetic to accommodate for an increased dataset size [52]. Concretely, for a dataset with only 212 entries and 14 features, the ring size should be at least 246 bits. Thus, it is necessary to redesign the protocols to enable computation within the standard ring sizes. For deep networks such as ResNet and LSTMs, they require performing batch normalization multiple times, each of which involves division and square-root operations [51]. Since the latter is expensive to perform over rings, designing efficient protocols for these operations is an interesting question.

Finally, although it is known how to instantiate the required primitives securely using standard MPC techniques, they are far from being practically efficient. Moreover, since the secure variant is known to have an overhead over the plaintext computation, sophisticated techniques are required to handle the large amount of intermediate data generated while training very deep networks. Existing PPML frameworks lack support for training the above ML algorithms to the best of our knowledge. We believe that accounting for the points above can bring the existing PPML frameworks, including Tetrad, one step closer to the efficient realization of these algorithms.

Acknowledgements

The authors would like to acknowledge support from Google PhD Fellowship 2019, Centre for Networked Intelligence (a Cisco CSR initiative) 2021, SERB MATRICS (Theoretical Sciences) Grant 2020 and Google India AI/ML Research Award 2020. The authors would also like to acknowledge the financial support from Google Cloud to perform the benchmarking.

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreements No. 850990 (PSOTI) and No. 803096 (SPEC)) and from the Digital Research Centre Denmark (DIREC).

References

[1] J. Alvarez-Valle, P. Bhatu, N. Chandran, D. Gupta, A. V. Nori, A. Rastogi, M. Rathee, R. Sharma, and S. Ugare, "Secure medical image analysis with cryptflow," *CoRR*, vol. abs/2012.05064, 2020. [Online]. Available: https://arxiv.org/abs/2012.05064 1

[2] M. Byali, H. Chaudhari, A. Patra, and A. Suresh, "FLASH: Fast and robust framework for privacy-preserving machine learning," *PoPETs*, vol. 2020, no. 2, pp. 459–480, Apr. 2020. 1, 3, 4

[3] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh, "ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction," in *ACM CCSW@CCS*, 2019. [Online]. Available: https://eprint.iacr.org/2019/429 1, 3, 4, 16, 21

[4] H. Chaudhari, R. Rachuri, and A. Suresh, "Trident: Efficient 4PC framework for privacy preserving machine learning," in *NDSS 2020*. The Internet Society, Feb. 2020. 1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 13, 16, 24

[5] E. Makri, D. Rotaru, N. P. Smart, and F. Vercauteren, "EPIC: Efficient private image classification (or: Learning from the masters)," in *CT-RSA 2019*, ser. LNCS, M. Matsui, Ed., vol. 11405. Springer, Heidelberg, Mar. 2019, pp. 473–492. 1

[6] P. Mohassel and P. Rindal, "ABY^3: A mixed protocol framework for machine learning," in *ACM CCS 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM Press, Oct. 2018, pp. 35–52. 1, 2, 3, 4, 6, 10, 11, 16, 21, 24

[7] P. Mohassel and Y. Zhang, "SecureML: A system for scalable privacy-preserving machine learning," in *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2017, pp. 19–38. 1, 2, 6, 11, 14, 21, 24

[8] A. Patra and A. Suresh, "BLAZE: Blazing fast privacy-preserving machine learning," in *NDSS 2020*. The Internet Society, Feb. 2020. 1, 2, 3, 4, 10, 11, 16, 24

[9] S. Wagh, D. Gupta, and N. Chandran, "SecureNN: 3-party secure computation for neural network training," *PoPETs*, vol. 2019, no. 3, pp. 26–49, Jul. 2019. 1, 2

[10] 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 18*, J. Kim, G.-J. Ahn, S. Kim, Y. Kim, J. López, and T. Kim, Eds. ACM Press, Apr. 2018, pp. 707–721. 1, 9, 11

[11] A. C.-C. Yao, "Protocols for secure computations (extended abstract)," in *23rd FOCS*. IEEE Computer Society Press, Nov. 1982, pp. 160–164. 1, 22

[12] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game or A completeness theorem for protocols with honest majority," in *19th ACM STOC*, A. Aho, Ed. ACM Press, May 1987, pp. 218–229. 1

[13] S. D. Gordon, S. Ranellucci, and X. Wang, "Secure computation with low communication from cross-checking," in *ASIACRYPT 2018, Part III*, ser. LNCS, T. Peyrin and S. Galbraith, Eds., vol. 11274. Springer, Heidelberg, Dec. 2018, pp. 59–85. 1, 16

[14] N. Koti, M. Pancholi, A. Patra, and A. Suresh, "SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning," in *USENIX Security'21*, 2021, https://eprint.iacr.org/2020/592. 1, 2, 4, 9, 10, 11, 13, 16, 18, 25

[15] I. Damgård, C. Orlandi, and M. Simkin, "Yet another compiler for active security or: Efficient MPC over arbitrary rings," in *CRYPTO 2018, Part II*, ser. LNCS, H. Shacham and A. Boldyreva, Eds., vol. 10992. Springer, Heidelberg, Aug. 2018, pp. 799–829. 1

[16] M. Keller, V. Pastro, and D. Rotaru, "Overdrive: Making SPDZ great again," in *EUROCRYPT 2018, Part III*, ser. LNCS, J. B. Nielsen and V. Rijmen, Eds., vol. 10822. Springer, Heidelberg, Apr. / May 2018, pp. 158–189. 1

[17] D. Demmler, T. Schneider, and M. Zohner, "ABY - A framework for efficient mixed-protocol secure two-party computation," in *NDSS 2015*. The Internet Society, Feb. 2015. 1, 16

[18] I. Damgård, D. Escudero, T. K. Frederiksen, M. Keller, P. Scholl, and N. Volgushev, "New primitives for actively-secure MPC over rings with applications to private machine learning," in *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2019, pp. 1102–1120. 1, 11, 25

[19] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara, "High-throughput semi-honest secure three-party computation with an honest majority," in *ACM CCS 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM Press, Oct. 2016, pp. 805–817. 1, 11, 16

[20] J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein, "High-throughput secure three-party computation for malicious adversaries and an honest majority," in *EUROCRYPT 2017, Part II*, ser. LNCS, J.-S. Coron and J. B. Nielsen, Eds., vol. 10211. Springer, Heidelberg, Apr. / May 2017, pp. 225–255. 1, 16

[21] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein, "Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier," in *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2017, pp. 843–862. 1, 11, 16

[22] M. Abspoel, A. Dalskov, D. Escudero, and A. Nof, "An efficient passive-to-active compiler for honest-majority MPC over rings," Cryptology ePrint Archive, Report 2019/1298, 2019, https://eprint.iacr.org/2019/1298. 1

[23] A. Patra, T. Schneider, A. Suresh, and H. Yalame, "ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation," in *USENIX Security'21*, 2021, https://eprint.iacr.org/2020/1225. 1, 2, 3, 9, 10, 16

[24] M. Byali, A. Joseph, A. Patra, and D. Ravi, "Fast secure computation for small population over the internet," in *ACM CCS 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM Press, Oct. 2018, pp. 677–694. 1, 3, 7, 16

[25] M. Byali, C. Hazay, A. Patra, and S. Singla, "Fast actively secure five-party computation with security beyond abort," in *ACM CCS 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM Press, Nov. 2019, pp. 1573–1590. 1

[26] D. Rotaru and T. Wood, "MArBled circuits: Mixing arithmetic and Boolean circuits with active security," in *INDOCRYPT 2019*, ser. LNCS, F. Hao, S. Ruj, and S. Sen Gupta, Eds., vol. 11898. Springer, Heidelberg, Dec. 2019, pp. 227–249. 1, 16

[27] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl, "Improved primitives for MPC over mixed arithmetic-binary circuits," in *CRYPTO 2020, Part II*, ser. LNCS, D. Micciancio and T. Ristenpart, Eds., vol. 12171. Springer, Heidelberg, Aug. 2020, pp. 823–852. 1, 16, 24

[28] S. Mazloom, P. H. Le, S. Ranellucci, and S. D. Gordon, "Secure parallel computation on national scale volumes of data," in *USENIX Security 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, Aug. 2020, pp. 2487–2504. 2, 16

[29] A. Dalskov, D. Escudero, and M. Keller, "Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security," in *USENIX Security'21*, 2021, https://eprint.iacr.org/2020/1330. 2, 13, 16, 17, 18

[30] S. Ohata and K. Nuida, "Communication-efficient (client-aided) secure two-party protocols and its application," in *FC 2020*, ser. LNCS, J. Bonneau and N. Heninger, Eds., vol. 12059. Springer, Heidelberg, Feb. 2020, pp. 369–385. 2

[31] Y. Ishai, R. Kumaresan, E. Kushilevitz, and A. Paskin-Cherniavsky, "Secure computation with minimal interaction, revisited," in *CRYPTO 2015, Part II*, ser. LNCS, R. Gennaro and M. J. B. Robshaw, Eds., vol. 9216. Springer, Heidelberg, Aug. 2015, pp. 359–378. 3, 7, 16

[32] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, pp. 2278–2324, 1998. 3, 11

[33] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014. 3, 11

[34] B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai, "SpOT-light: Lightweight private set intersection from sparse OT extension," in *CRYPTO 2019, Part III*, ser. LNCS, A. Boldyreva and D. Micciancio, Eds., vol. 11694. Springer, Heidelberg, Aug. 2019, pp. 401–431. 3

[35] M. Fredrikson, S. Jha, and T. Ristenpart, "Model inversion attacks that exploit confidence information and basic countermeasures," in *ACM CCS 2015*, I. Ray, N. Li, and C. Kruegel, Eds. ACM Press, Oct. 2015, pp. 1322–1333. 3

[36] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction APIs," in *USENIX Security 2016*, T. Holz and S. Savage, Eds. USENIX Association, Aug. 2016, pp. 601–618. 3

[37] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2017, pp. 3–18. 3

[38] M. Abspoel, R. Cramer, I. Damgård, D. Escudero, and C. Yuan, "Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings," in *TCC 2019, Part I*, ser. LNCS, D. Hofheinz and A. Rosen, Eds., vol. 11891. Springer, Heidelberg, Dec. 2019, pp. 471–501. 7

[39] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof, "Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs," in *ACM CCS 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM Press, Nov. 2019, pp. 869–886. 7, 16

[40] P. Mohassel, M. Rosulek, and Y. Zhang, "Fast and secure three-party computation: The garbled circuit approach," in *ACM CCS 2015*, I. Ray, N. Li, and C. Kruegel, Eds. ACM Press, Oct. 2015, pp. 591–602. 7, 16

[41] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free XOR gates and applications," in *ICALP 2008, Part II*, ser. LNCS, L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, Eds., vol. 5126. Springer, Heidelberg, Jul. 2008, pp. 486–498. 8

[42] V. Kolesnikov, P. Mohassel, and M. Rosulek, "FleXOR: Flexible garbling for XOR gates that beats free-XOR," in *CRYPTO 2014, Part II*, ser. LNCS, J. A. Garay and R. Gennaro, Eds., vol. 8617. Springer, Heidelberg, Aug. 2014, pp. 440–457. 8

[43] S. Zahur, M. Rosulek, and D. Evans, "Two halves make a whole - reducing data transfer in garbled circuits using half gates," in *EUROCRYPT 2015, Part II*, ser. LNCS, E. Oswald and M. Fischlin, Eds., vol. 9057. Springer, Heidelberg, Apr. 2015, pp. 220–250. 8

[44] S. Gueron, Y. Lindell, A. Nof, and B. Pinkas, "Fast garbling of circuits under standard assumptions," *Journal of Cryptology*, vol. 31, no. 3, pp. 798–844, Jul. 2018. 8

[45] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient garbling from a fixed-key blockcipher," in *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2013, pp. 478–492. 8

[46] Stanford, "CS231n: Convolutional Neural Networks for Visual Recognition," https://cs231n.github.io/convolutional-networks/. 10

[47] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/ 11, 12

[48] A. Krizhevsky, V. Nair, and G. Hinton, "The CIFAR-10 dataset," 2014, https://www.cs.toronto.edu/~kriz/cifar.html. 11, 12

[49] Cryptography and P. E. G. at TU Darmstadt, "ENCRYPTO Utils," https://github.com/encryptogroup/ENCRYPTO_utils, 2017. 11

[50] P. Miao, S. Patel, M. Raykova, K. Seth, and M. Yung, "Two-sided malicious security for private intersection-sum with cardinality," in *CRYPTO 2020, Part III*, ser. LNCS, D. Micciancio and T. Ristenpart, Eds., vol. 12172. Springer, Heidelberg, Aug. 2020, pp. 3–33. 11

[51] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin, "Falcon: Honest-majority maliciously secure framework for private deep learning," *PoPETs*, vol. 2021, no. 1, pp. 188–208, Jan. 2021. 11, 14, 16, 24

[52] D. Cabarcas, H. D. Vanegas, and D. E. Escudero, "Privacy-preserving machine learning for support vector machines," Privacy-Preserving Machine Learning Workshop (PPML@CRYPTO'21), 2021. 14

[53] 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, Part III*, ser. LNCS, A. Boldyreva and D. Micciancio, Eds., vol. 11694. Springer, Heidelberg, Aug. 2019, pp. 67–97. 16

[54] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, "TASTY: tool for automating secure two-party computations," in *ACM CCS 2010*, E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, Eds. ACM Press, Oct. 2010, pp. 451–462. 16

[55] P. Rogaway and T. Shrimpton, "Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance," in *FSE 2004*, ser. LNCS, B. K. Roy and W. Meier, Eds., vol. 3017. Springer, Heidelberg, Feb. 2004, pp. 371–388. 17

[56] O. Goldreich, *Foundations of Cryptography: Basic Applications*. Cambridge, UK: Cambridge University Press, 2004, vol. 2. 17

[57] Y. Lindell, "How to simulate it - A tutorial on the simulation proof technique," Cryptology ePrint Archive, Report 2016/046, 2016, https://eprint.iacr.org/2016/046. 17

[58] B. Alon, E. Omri, and A. Paskin-Cherniavsky, "MPC with friends and foes," in *CRYPTO 2020, Part II*, ser. LNCS, D. Micciancio and T. Ristenpart, Eds., vol. 12171. Springer, Heidelberg, Aug. 2020, pp. 677–706. 17

[59] M. Bellare, V. T. Hoang, and P. Rogaway, "Foundations of garbled circuits," in *ACM CCS 2012*, T. Yu, G. Danezis, and V. D. Gligor, Eds. ACM Press, Oct. 2012, pp. 784–796. 22

[60] P. Pullonen and S. Siim, "Combining secret sharing and garbled circuits for efficient private IEEE 754 floating-point computations," in *FC 2015 Workshops*, ser. LNCS, M. Brenner, N. Christin, B. Johnson, and K. Rohloff, Eds., vol. 8976. Springer, Heidelberg, Jan. 2015, pp. 172–183. 24

APPENDIX A
PRELIMINARIES

A. Related Work

Related work covers MPC protocols with an honest majority for high-throughput and constant-round setting and mixed-protocol frameworks for the case of PPML.

ABY3 [6] was the first framework for the case of 3 parties, supporting both training and inference. It had variants for both passive and active security, with the former being based on [19] and the latter on [20], [21]. ASTRA [3] improved upon the 3PC of [19]–[21] by proposing faster protocols for the online phase with active security. As a result, secure inference of ASTRA is faster than ABY3. Building on [53], BLAZE [8] proposed an actively secure framework that supports inference of neural networks. BLAZE pushes the expensive zero-knowledge part of the computation to the preprocessing phase, making its online phase faster than that of [53]. SWIFT (3PC) improved upon BLAZE by using the distributed zero-knowledge protocol of [39], thereby achieving GOD. In an orthogonal line of work, FALCON [51] focused on enhancing the efficiency of actively secure protocols for large convolutional neural networks, supporting training and inference.

In the high-throughput setting for 4PC, [13] explores protocols for the security notions of abort. Inspired by the theoretical GOD construction in [13], FLASH proposed practical protocols with GOD for secure inference. Trident [4] improved protocols (in terms of communication) compared to [13] with a focus on security with fairness. In addition, it was the first work to propose a mixed-protocol framework for the case of 4 parties. More recently, [28] improved over [13] to provide support for fixed-point arithmetic with applications to graph parallel computation, albeit with abort security.

Improving the security of Trident to GOD, SWIFT [14] presented an efficient, robust PPML framework with protocols as fast as Trident. SWIFT only supports the secure inference of neural networks and lacks conversions similar to the ones from Trident and the garbled world. Fantastic Four [29] also provides robust 4PC protocols which are on par with SWIFT. While they claim to provide a better security model called *private robustness* compared to SWIFT, it has been shown in SWIFT that the two security models are theoretically equivalent. Our security model is also similar to SWIFT, and we elaborate on its equivalence to private robustness in §A-C.

In the regime of constant-round protocols, [40] presents 3PC protocols in the honest majority setting satisfying security with abort, which require communicating one garbled circuit and three rounds of interaction. The work of [31] presents a robust 4-party computation protocol (4PC) with GOD in 2-rounds (which is optimal) at the expense of 12 garbled circuits. Further, [24] presents efficient 3PC and 4PC constructions providing security notions of fairness and GOD.

A mixed-protocol framework for MPC was first shown to be practical, in the 2-party dishonest majority setting, by TASTY [54]. TASTY was a passively secure compiler supporting generation of protocols based on homomorphic encryption and garbled circuits. This was followed by ABY [17], which proposed a mixed protocol framework, also with passive security, combining the arithmetic, boolean and garbled worlds. The recent work of ABY2 [23] improves upon the ABY framework, providing a faster online phase with applications to PPML. The work of [26], [27] proposed efficient mixed world conversions for the case of n parties with a dishonest majority. Both works have active security, with [26] supporting the inference of SVMs, and [27] supporting neural network inference.

In the honest majority setting, ABY3 [6] extended the idea to 3 parties and provided specialised protocols for the case of PPML. ABY3 was the first work to support secure training in the case of 3 parties, while Trident [4] extended it to the 4-party setting.

B. Basic Primitives

a) Shared Key Setup: Let $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \to X$ be a secure pseudo-random function (PRF), with co-domain X

being $\mathbb{Z}_{2^\ell}$. The following set of keys are established between the parties.

– One key between every pair – k_{ij} for P_i, P_j.
– One key between every set of three parties – k_{ijk} for P_i, P_j, P_k.
– One shared keys $k_{\mathcal{P}}$ known to all parties in $\mathcal{P}$.

Suppose P_0, P_1 wish to sample a random value $r \in \mathbb{Z}_{2^\ell}$ non-interactively. To do so they invoke $F_{k_{01}}(id_{01})$ and obtain r. Here, id_{01} denotes a counter maintained by the parties, and is updated after every PRF invocation. The appropriate keys used to sample is implicit from the context, from the identities of the pair that sample or from the fact that it is sampled by all, and, hence, is omitted.

Functionality $\mathcal{F}_{\text{SETUP}}$

$\mathcal{F}_{\text{SETUP}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\text{SETUP}}$ picks random keys k_{ij} and k_{ijk} for $i, j, k \in \{0, 1, 2, 3\}$ and $k_{\mathcal{P}}$. Let y_s denote the keys corresponding to party P_s. Then

– $\mathsf{y}_s = (k_{01}, k_{02}, k_{03}, k_{012}, k_{013}, k_{023}$ and $k_{\mathcal{P}})$ when $P_s = P_0$.
– $\mathsf{y}_s = (k_{01}, k_{12}, k_{13}, k_{012}, k_{013}, k_{123}$ and $k_{\mathcal{P}})$ when $P_s = P_1$.
– $\mathsf{y}_s = (k_{02}, k_{12}, k_{23}, k_{012}, k_{023}, k_{123}$ and $k_{\mathcal{P}})$ when $P_s = P_2$.
– $\mathsf{y}_s = (k_{03}, k_{13}, k_{23}, k_{013}, k_{023}, k_{123}$ and $k_{\mathcal{P}})$ when $P_s = P_3$.

Output: Send $(\mathsf{Output}, \mathsf{y}_s)$ to every $P_s \in \mathcal{P}$.

Figure 9: Ideal functionality for shared-key setup

The key setup is modelled via a functionality $\mathcal{F}_{\text{SETUP}}$ (Fig. 9) that can be realised using any secure MPC protocol. A simple instantiation of such an MPC protocol is as follows. P_i samples key k_{ij} and sends to P_j. P_i samples k_{ijk} and sens to P_j. P_i, P_j jsnd k_{ijk} to P_k. Similarly, P_0 samples $k_{\mathcal{P}}$ and sends to P_3. P_0, P_3 jsnd $k_{\mathcal{P}}$ to P_1 and P_2.

b) Collision-Resistant Hash Function [55]: . A family of hash functions $\{\mathsf{H} : \mathcal{K} \times \mathsf{M} \rightarrow \mathcal{Y}\}$ is said to be collision resistant if for all PPT adversaries $\mathcal{A}$, given the hash function H_k for $k \in_R \mathcal{K}$, the following holds: $\Pr[(x, x') \leftarrow \mathcal{A}(k) : (x \neq x') \wedge \mathsf{H}_k(x) = \mathsf{H}_k(x')] = \mathsf{negl}(\kappa)$, where $x, x' \in \{0, 1\}^m$ and $m = \mathsf{poly}(\kappa)$.

C. Security Model

We prove security using the real-world/ ideal-word simulation paradigm [56], [57]. The security is analyzed by comparing what an adversary can do in the real world's execution of the protocol with what it can do in an ideal world execution where there is a trusted third party and is considered secure by definition. In the ideal world, the parties send their inputs to the trusted third party over perfectly secure channels that carries out the computation and sends the output to the parties. Informally, a protocol is secure if whatever an adversary can do in the real world can also be done in the ideal world.

Let $\mathcal{A}$ denote the probabilistic polynomial time (PPT) real-world adversary corrupting at most one party in $\mathcal{P}$, $\mathcal{S}$ denote the corresponding ideal world adversary, and $\mathcal{F}$ denote the ideal functionality. Let $\text{IDEAL}_{\mathcal{F},\mathcal{S}}(1^\kappa, z)$ denote the joint output of the honest parties and $\mathcal{S}$ from the ideal execution with respect to the security parameter κ and auxiliary input z. Similarly, let $\text{REAL}_{\Pi,\mathcal{A}}(1^\kappa, z)$ denote the joint output of the honest parties and $\mathcal{A}$ from the real world execution. We say that the protocol Π securely realizes $\mathcal{F}$ if for every PPT adversary $\mathcal{A}$ there exists an ideal world adversary $\mathcal{S}$ corrupting the same parties such that $\text{IDEAL}_{\mathcal{F},\mathcal{S}}(1^\kappa, z)$ and $\text{REAL}_{\Pi,\mathcal{A}}(1^\kappa, z)$ are computationally indistinguishable. The ideal functionality for computing a function f with fairness and robustness appears in Fig. 10 and Fig. 11, respectively.

Functionality $\mathcal{F}_{\text{FAIR}}$

Every honest party $P_i \in \mathcal{P}$ sends its input x_i to the functionality. Corrupted parties may send arbitrary inputs as instructed by the adversary. While sending the inputs, the adversary is also allowed to send a special `abort` command.

Input: On message (Input, x_i) from P_i, do the following: if $(\mathsf{Input}, *)$ already received from P_i, then ignore the current message. Otherwise, record $x'_i = x_i$ internally. If x_i is outside P_i's domain, consider $x'_i =$ `abort`.

Output: If there exists an $i \in \{0, 1, 2, 3\}$ such that $x'_i =$ `abort`, send $(\mathsf{Output}, \perp)$ to all the parties. Else, compute $y = f(x'_0, x'_1, x'_2, x'_3)$ and send (Output, y) to all parties.

Figure 10: Fair functionality for computing function f

Functionality $\mathcal{F}_{\text{ROBUST}}$

Every honest party $P_i \in \mathcal{P}$ sends its input x_i to the functionality. Corrupted parties may send arbitrary inputs as instructed by the adversary.

Input: On message (Input, x_i) from P_i, do the following: if $(\mathsf{Input}, *)$ already received from P_i, then ignore the current message. Otherwise, record $x'_i = x_i$ internally. If x_i is outside P_i's domain, consider x'_i to be some predetermined default value.

Output: Compute $y = f(x'_0, x'_1, x'_2, x'_3)$ and send (Output, y) to all parties.

Figure 11: Robust functionality for computing function f

a) On the security of robust Tetrad*:* We emphasize that we follow the standard traditional (real-world / ideal-world based) security definition of MPC, according to which, in the 4-party setting with one corruption, exactly one party is assumed to be corrupt, and the rest are *honest*. As per this definition, disclosing the honest parties' inputs to a selected *honest* party is *not* a breach of security. Indeed in Tetrad, the data sharing and the computation on the shared data are done so that any malicious behaviour leads to establishing a trusted party P_{TP} who is enabled to receive all the inputs and compute the output on the clear. There has been a recent study on the additional requirement of hiding the inputs from a quorum of honest parties (treating them as semi-honest), termed as Friends-and-Foes (FaF) security notion [58]. This is a stronger security goal than the standard one. Informally, designing secure 4PC FaF protocols requires security against two independent corruptions. Our sharing semantics, designed to handle only one corruption, does not suffice. Hence, we leave FaF-secure 4PC for future exploration.

Another security notion, called *private robustness*, was recently proposed in the work of Dalskov et al. [29], where the protocol does not demand the inputs be sent to a P_{TP}. Their work, however, considers a more restricted security model, where it is assumed that parties will discard messages which are *non-intended* and are not a part of the protocol. This involves assuming a *secure erasure*. Under this assumption, our model is equivalent to that of private robustness since the trusted party $\mathsf{P_{TP}}$ will erase the input of the honest parties after computing the function output.

D. Comparison with Fantastic Four [29]

We analyse the performance of Fantastic Four [29] where execution proceeds in segments (cf. §6.4, [29]). Elaborately, computation is carried out optimistically for each segment, followed by a verification phase before proceeding to the next segment. If verification fails, the current segment is recomputed via an active 3PC protocol. Subsequent segments also proceed with a 3PC execution until the verification fails again. In this case, a semi-honest 2PC with a helper is carried out for the current and rest of the segments. For analysis, we consider their best and worst-case execution cost.

Protocol	Dot Product w/ Truncation		#Active Parties
	Preprocessing	Online	
Fantastic Four: Case I	ℓ	9ℓ	4
Fantastic Four: Case II	$76(\ell+\kappa)+54x+12$	$9\ell+6\kappa$	3
Tetrad-R(on-demand)	-	5ℓ	3

Table X: Comparison with Fantastic Four [29]

Observe that the best case happens when the verification is always successful, which we call as *Case I*. In this case, the communication cost is that of the 4PC execution. Note that an adversary can *always* make the verification fail in the first segment itself. This results in executing the entire protocol (all segments) with their active 3PC, which accounts for their worst-case cost. We denote this as *Case II*. Their 3PC protocols are designed to work over the extended ring of size $\ell+\kappa$ bits. As evident from Tables 2, 3 of their paper, their 3PC is at least $10\times$ more expensive than their 4PC in terms of both runtime and communication. Thus, the higher cost of 3PC defeats the purpose of having an additional honest party in the system.

Observe that their protocols are designed to work with a function-independent preprocessing. Thus, for a fair comparison, we compare both cases against the on-demand variant of our robust protocols (Tetrad-R). The results are summarised in Table X. We remark that the values for their cases are obtained from Table 1 of their paper [29].

Appendix B
4PC Protocol

Here we detail the additional information regarding the 4PC protocols.

a) Joint-send for robust protocols: The formal protocol for Π_{jsnd} in the robust setting [14] is given in Fig. 12.

Lemma B.1 (Communication)**.** *Protocol Π_{jsnd} (Fig. 12) requires an amortized communication of ℓ bits and 1 round.*

Proof: In the protocol $\Pi_{\mathsf{jsnd}}(P_i, P_j, \mathsf{v}, P_k)$ for the fair variant, P_i communicates v to P_k requiring communication of ℓ bits and one round. The hash value communication from P_j to P_k can be clubbed for multiple instances with the same set of parties and hence the cost gets amortized. The analysis is similar for the robust case as well. Here, though the verification consists of multiple steps, the cost gets amortized over multiple instances. ■

Protocol $\Pi_{\mathsf{jsnd}}(P_i, P_j, \mathsf{v}, P_k)$

$P_s \in \mathcal{P}$ initializes an inconsistency bit $\mathsf{b}_s = 0$. If P_s remains silent instead of sending b_s in any of the following rounds, the recipient sets b_s to 1.

– *Send:* P_i sends v to P_k.
– *Verify:* P_j sends $\mathsf{H}(\mathsf{v})$ to P_k.
- P_k sets $\mathsf{b}_k = 1$ if the received values are inconsistent or if the value is not received.
- P_k sends b_k to all parties. P_s for $s \in \{i,j,l\}$ sets $\mathsf{b}_s = \mathsf{b}_k$.
- P_s for $s \in \{i,j,l\}$ mutually exchange their bits. P_s resets $\mathsf{b}_s = \mathsf{b}'$ where b' denotes the bit which appears in majority among $\mathsf{b}_i, \mathsf{b}_j, \mathsf{b}_l$.
- All parties set $\mathsf{P_{TP}} = P_l$ if $\mathsf{b}' = 1$, terminate otherwise.

Figure 12: Joint-Send for robust protocols

b) Sharing Protocol:

Lemma B.2 (Communication)**.** *Protocol Π_{Sh} (Fig. 1) requires an amortized communication of at most 3ℓ bits and 1 round in the online phase.*

Proof: The preprocessing of Π_{Sh} is non-interactive as the parties sample non interactively using key setup $\mathcal{F}_{\text{SETUP}}$ (§A-B). in the online phase, P_i sends $\mathsf{m_v}$ to P_1, P_2, P_3 resulting in 1 round and communication of at most 3ℓ bits ($P_i = P_0$). The next round of hash exchange can be clubbed for several instances and the cost gets amortized over multiple instances. ■

c) Reconstruction Protocol:

Lemma B.3 (Communication)**.** *Protocol Π_{Rec} (Fig. 13) requires an amortized communication of 4ℓ bits and 1 round in the online phase.*

Proof: The protocol involves 4 invocations of Π_{jsnd} protocol and the communication follows from Lemma B.1. ■

Protocol $\Pi_{\mathsf{Rec}}(\mathcal{P}, [\![\mathsf{v}]\!])$

1) P_1, P_0 jsnd λ_{v}^1 to P_2; P_2, P_0 jsnd λ_{v}^3 to P_3; P_3, P_0 jsnd λ_{v}^2 to P_1; P_1, P_2 jsnd $\mathsf{m_v}$ to P_0.
2) Compute $\mathsf{v} = \mathsf{m_v} - \lambda_{\mathsf{v}}^1 - \lambda_{\mathsf{v}}^2 - \lambda_{\mathsf{v}}^3$.

Figure 13: Reconstruction (with abort) of v among $\mathcal{P}$.

d) Multiplication Protocol:

Lemma B.4 (Communication)**.** *Protocol Π_{Mult} (Fig. 3) (in Tetrad) requires 2ℓ bits of communication in the preprocessing phase, and 1 round and 3ℓ bits of communication in the online phase.*

Proof: During preprocessing, sampling of values $\mathsf{u}^1, \mathsf{u}^2$ are performed non-interactively using $\mathcal{F}_{\text{SETUP}}$. A communication of ℓ bits is required for the joint sharing of q by P_0, P_3 as explained in §III-A0d. In addition, P_0 communicates w to P_3 requiring additional ℓ bits. During online, two instances of Π_{jsnd} are executed in parallel resulting in a communication of 2ℓ bits and 1 round. This is followed by a joint sharing by P_1, P_2 to P_3 for which an additional communication of ℓ bits are required. However, in joint sharing, the communication is from P_1 to P_3 and the same can be deferred till the verification stage. Thus the online round is retained as 1 in an amortized sense. ∎

Lemma B.5 (Communication)**.** *Protocol Π_{Mult} (Fig. 3) (in Tetrad-R) requires 2ℓ bits of communication in the preprocessing phase, and 1 round and 3ℓ bits of communication in the online phase.*

A. Function-independent preprocessing

We provide the fair multiplication, $\Pi^{\mathsf{NoPre}}_{\mathsf{Mult}}$, for *function-independent* preprocessing in Fig. 14. The protocol incurs no overhead over the fair multiplication (Π_{Mult}) in Tetrad. This is due to the design of Π_{Mult} where values $\mathsf{u}^1, \mathsf{u}^2$ are sampled non-interactively in the preprocessing. Thus the joint-sharing by P_0, P_3 (Step 5 (a) in Fig. 14) can be performed along with the communication among P_1, P_2 (Step 4 in Fig. 14) in the online. Moreover, the rest of the communication can be deferred till the verification stage and thus, the online round complexity is retained. The protocol for robust setting is similar.

Protocol $\Pi^{\mathsf{NoPre}}_{\mathsf{Mult}}(\mathsf{a}, \mathsf{b}, \mathsf{isTr})$

Let isTr be a bit that denotes whether truncation is required ($\mathsf{isTr} = 1$) or not ($\mathsf{isTr} = 0$).

Online:

1) Locally compute the following:

$$P_0, P_1 : \gamma^1_{\mathsf{ab}} = \lambda^1_{\mathsf{a}}\lambda^3_{\mathsf{b}} + \lambda^3_{\mathsf{a}}\lambda^1_{\mathsf{b}} + \lambda^3_{\mathsf{a}}\lambda^3_{\mathsf{b}}$$
$$P_0, P_2 : \gamma^2_{\mathsf{ab}} = \lambda^2_{\mathsf{a}}\lambda^3_{\mathsf{b}} + \lambda^3_{\mathsf{a}}\lambda^2_{\mathsf{b}} + \lambda^2_{\mathsf{a}}\lambda^2_{\mathsf{b}}$$
$$P_0, P_3 : \gamma^3_{\mathsf{ab}} = \lambda^1_{\mathsf{a}}\lambda^2_{\mathsf{b}} + \lambda^2_{\mathsf{a}}\lambda^1_{\mathsf{b}} + \lambda^1_{\mathsf{a}}\lambda^1_{\mathsf{b}}$$

2) P_0, P_3 and P_j sample random $\mathsf{u}^j \in_R \mathbb{Z}_{2^\ell}$ for $j \in \{1, 2\}$. Let $\mathsf{u}^1 + \mathsf{u}^2 = \gamma^3_{\mathsf{ab}} - \mathsf{r}$ for a random $\mathsf{r} \in_R \mathbb{Z}_{2^\ell}$.
3) Let $\mathsf{y} = (\mathsf{z} - \mathsf{r}) - \mathsf{m_a m_b}$. Locally compute the following:

$$P_1 : \mathsf{y}_1 = -\lambda^1_{\mathsf{a}}\mathsf{m_b} - \lambda^1_{\mathsf{b}}\mathsf{m_a} + \gamma^1_{\mathsf{ab}} + \mathsf{u}^1$$
$$P_2 : \mathsf{y}_2 = -\lambda^2_{\mathsf{a}}\mathsf{m_b} - \lambda^2_{\mathsf{b}}\mathsf{m_a} + \gamma^2_{\mathsf{ab}} + \mathsf{u}^2$$
$$P_1, P_2 : \mathsf{y}_3 = -\lambda^3_{\mathsf{a}}\mathsf{m_b} - \lambda^3_{\mathsf{b}}\mathsf{m_a}$$

4) P_1 sends y_1 to P_2, while P_2 sends y_2 to P_1.
5) Parties proceed as follows:
   a) P_0, P_3: $\mathsf{r} = \gamma^3_{\mathsf{ab}} - \mathsf{u}^1 - \mathsf{u}^2$; $\mathsf{q} = \mathsf{r}^t$ if $\mathsf{isTr} = 1$, else $\mathsf{q} = \mathsf{r}$; Execute $\Pi_{\mathsf{JSh}}(P_0, P_3, \mathsf{q})$.
   b) P_1, P_2: $\mathsf{z} - \mathsf{r} = (\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{y}_3) + \mathsf{m_a m_b}$; $\mathsf{p} = (\mathsf{z} - \mathsf{r})^t$ if $\mathsf{isTr} = 1$, else $\mathsf{p} = \mathsf{z} - \mathsf{r}$; Execute $\Pi_{\mathsf{JSh}}(P_1, P_2, \mathsf{p})$.
6) Locally compute $[\![\mathsf{o}]\!] = [\![\mathsf{p}]\!] + [\![\mathsf{q}]\!]$. Here $\mathsf{o} = \mathsf{z}^t$ if $\mathsf{isTr} = 1$ and z otherwise.

Verification:

1) P_0, P_1, P_2 sample random $\mathsf{s}_1, \mathsf{s}_2 \in_R \mathbb{Z}_{2^\ell}$ and set $\mathsf{s} = \mathsf{s}_1 + \mathsf{s}_2$. P_0 sends $\mathsf{w} = \gamma^1_{\mathsf{ab}} + \gamma^2_{\mathsf{ab}} + \mathsf{s}$ to P_3.
2) P_3 computes $\mathsf{v} = -(\lambda^1_{\mathsf{a}} + \lambda^2_{\mathsf{a}})\mathsf{m_b} - (\lambda^1_{\mathsf{b}} + \lambda^2_{\mathsf{b}})\mathsf{m_a} + \mathsf{u}^1 + \mathsf{u}^2 + \mathsf{w}$ and sends $\mathsf{H}(\mathsf{v})$ to P_1 and P_2. Parties P_1, P_2 `abort` iff $\mathsf{H}(\mathsf{v}) \neq \mathsf{H}(\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{s})$.

Figure 14: Fair multiplication without preprocessing.

Appendix C
Building Blocks

a) Dot Product (Scalar Product):

Lemma C.1 (Communication)**.** *Protocol Π_{dotp} (Fig. 6) (in Tetrad) requires 2ℓ bits of communication in preprocessing, and 1 round and 3ℓ bits of communication in the online phase.*

Proof: Here, the parties add up the locally computed shares corresponding to each partial product of the form $\mathsf{a}_i\mathsf{b}_i$ and then performs the communication of the sum. The communication pattern is similar to that of the fair multiplication protocol (Fig. 3) and the costs follow from Lemma B.4. ∎

b) Multi-input Multiplication:

Lemma C.2 (Communication)**.** *Protocol Π_{Mult3} (Fig. 15) (in Tetrad) requires 9ℓ bits of communication in preprocessing, and 1 round and 3ℓ bits of communication in the online phase.*

Proof: In the preprocessing, computation of γ_{ab} involves three instances of jsnd. Each of the computation of $\gamma_{\mathsf{ac}}, \gamma_{\mathsf{bc}}$ involves one instance of jsnd and a communication from P_0 to P_3. The computation of γ_{abc} is similar to the preprocessing of fair multiplication protocol (Fig. 3). The communication pattern of the online phase is similar to that of the fair multiplication protocol. The costs follow from Lemma B.4 and Lemma B.1. ∎

For the robust 3-input multiplication, correctness of three messages, $\mathsf{w_{ac}}, \mathsf{w_{bc}}, \mathsf{w_{abc}}$, sent by P_0 have to be verified by invoking Π_{VrfyP0}.

Protocol $\Pi_{\mathsf{Mult3}}(\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{isTr})$

Let isTr be a bit that denotes whether truncation is required ($\mathsf{isTr} = 1$) or not ($\mathsf{isTr} = 0$).

Preprocessing:

1) Computation for γ_{ab}: Invoke $\Pi_{\mathsf{MulR}}(\lambda_{\mathsf{a}}, \lambda_{\mathsf{b}})$ (Fig. 2).
2) Computation for γ_{ac}:
   – Locally compute the following:

$$P_0, P_1 : \gamma^1_{\mathsf{ac}} = \lambda^1_{\mathsf{a}}\lambda^3_{\mathsf{c}} + \lambda^3_{\mathsf{a}}\lambda^1_{\mathsf{c}} + \lambda^3_{\mathsf{a}}\lambda^3_{\mathsf{c}}$$
$$P_0, P_2 : \gamma^2_{\mathsf{ac}} = \lambda^2_{\mathsf{a}}\lambda^3_{\mathsf{c}} + \lambda^3_{\mathsf{a}}\lambda^2_{\mathsf{c}} + \lambda^2_{\mathsf{a}}\lambda^2_{\mathsf{c}}$$
$$P_0, P_3 : \gamma^3_{\mathsf{ac}} = \lambda^1_{\mathsf{a}}\lambda^2_{\mathsf{c}} + \lambda^2_{\mathsf{a}}\lambda^1_{\mathsf{c}} + \lambda^1_{\mathsf{a}}\lambda^1_{\mathsf{c}}$$

   – P_0, P_3 and P_1 sample random $\mathsf{u}^1_{\mathsf{ac}} \in_R \mathbb{Z}_{2^\ell}$. P_0, P_3 compute and jsnd $\mathsf{u}^2_{\mathsf{ac}} = \gamma^3_{\mathsf{ac}} - \mathsf{u}^1_{\mathsf{ac}}$ to P_2.
   – P_0, P_1, P_2 sample random $\mathsf{s}_{\mathsf{ac}_1}, \mathsf{s}_{\mathsf{ac}_2} \in_R \mathbb{Z}_{2^\ell}$ and set $\mathsf{s_{ac}} = \mathsf{s}_{\mathsf{ac}_1} + \mathsf{s}_{\mathsf{ac}_2}$. P_0 sends $\mathsf{w_{ac}} = \gamma^1_{\mathsf{ac}} + \gamma^2_{\mathsf{ac}} + \mathsf{s_{ac}}$ to P_3.
3) Computation for γ_{bc}: Similar to Step 2 (for γ_{ac}). P_1, P_2 obtain $\mathsf{u}^1_{\mathsf{bc}}, \mathsf{u}^2_{\mathsf{bc}}$ respectively such that $\mathsf{u}^1_{\mathsf{bc}} + \mathsf{u}^2_{\mathsf{bc}} = \gamma^3_{\mathsf{bc}}$. P_3 obtains $\mathsf{w_{bc}} = \gamma^1_{\mathsf{bc}} + \gamma^2_{\mathsf{bc}} + \mathsf{s_{bc}}$.

4) Computation for γ_{abc}:
   – Using γ_{ab} (Step 1), λ_{c}, compute the following:

$$P_0, P_1 : \gamma^1_{\mathsf{abc}} = \gamma^1_{\mathsf{ab}}\lambda^3_{\mathsf{c}} + \gamma^3_{\mathsf{ab}}\lambda^1_{\mathsf{c}} + \gamma^3_{\mathsf{ab}}\lambda^3_{\mathsf{c}}$$
$$P_0, P_2 : \gamma^2_{\mathsf{abc}} = \gamma^2_{\mathsf{ab}}\lambda^3_{\mathsf{c}} + \gamma^3_{\mathsf{ab}}\lambda^2_{\mathsf{c}} + \gamma^2_{\mathsf{ab}}\lambda^2_{\mathsf{c}}$$
$$P_0, P_3 : \gamma^3_{\mathsf{abc}} = \gamma^1_{\mathsf{ab}}\lambda^2_{\mathsf{c}} + \gamma^2_{\mathsf{ab}}\lambda^1_{\mathsf{c}} + \gamma^1_{\mathsf{ab}}\lambda^1_{\mathsf{c}}$$

   – P_0, P_3 and P_j sample random $\mathsf{u}^j_{\mathsf{abc}} \in_R \mathbb{Z}_{2^\ell}$ for $j \in \{1, 2\}$. Let $\mathsf{u}^1_{\mathsf{abc}} + \mathsf{u}^2_{\mathsf{abc}} = \gamma^3_{\mathsf{abc}} + \mathsf{r}$ for $\mathsf{r} \in_R \mathbb{Z}_{2^\ell}$.
   – P_0, P_1, P_2 sample random $\mathsf{s}_1, \mathsf{s}_2 \in_R \mathbb{Z}_{2^\ell}$ and set $\mathsf{s} = \mathsf{s}_1 + \mathsf{s}_2$[a]. P_0 sends $\mathsf{w}_{\mathsf{abc}} = \gamma^1_{\mathsf{abc}} + \gamma^2_{\mathsf{abc}} + \mathsf{s}$ to P_3.

5) P_0, P_3 compute $\mathsf{r} = \mathsf{u}^1_{\mathsf{abc}} + \mathsf{u}^2_{\mathsf{abc}} - \gamma^3_{\mathsf{abc}}$ and set $\mathsf{q} = \mathsf{r}^t$ if $\mathsf{isTr} = 1$, else set $\mathsf{q} = \mathsf{r}$. Execute $\Pi_{\mathsf{JSh}}(P_0, P_3, \mathsf{q})$ to generate $[\![\mathsf{q}]\!]$.

Online: Let $\mathsf{y} = (\mathsf{z} - \mathsf{r}) - \mathsf{m}_{\mathsf{abc}}$.

1) Locally compute the following:

$$\begin{aligned} P_1 : \mathsf{y}_1 &= -\lambda^1_{\mathsf{a}}\mathsf{m}_{\mathsf{bc}} - \lambda^1_{\mathsf{b}}\mathsf{m}_{\mathsf{ac}} - \lambda^1_{\mathsf{c}}\mathsf{m}_{\mathsf{ab}} + \gamma^1_{\mathsf{ab}}\mathsf{m}_{\mathsf{c}} \\ &+ (\gamma^1_{\mathsf{ac}} + \mathsf{u}^1_{\mathsf{ac}})\mathsf{m}_{\mathsf{b}} + (\gamma^1_{\mathsf{bc}} + \mathsf{u}^1_{\mathsf{bc}})\mathsf{m}_{\mathsf{a}} - (\gamma^1_{\mathsf{abc}} + \mathsf{u}^1_{\mathsf{abc}}) \\ P_2 : \mathsf{y}_2 &= -\lambda^2_{\mathsf{a}}\mathsf{m}_{\mathsf{bc}} - \lambda^2_{\mathsf{b}}\mathsf{m}_{\mathsf{ac}} - \lambda^2_{\mathsf{c}}\mathsf{m}_{\mathsf{ab}} + \gamma^2_{\mathsf{ab}}\mathsf{m}_{\mathsf{c}} \\ &+ (\gamma^2_{\mathsf{ac}} + \mathsf{u}^2_{\mathsf{ac}})\mathsf{m}_{\mathsf{b}} + (\gamma^2_{\mathsf{bc}} + \mathsf{u}^2_{\mathsf{bc}})\mathsf{m}_{\mathsf{a}} - (\gamma^2_{\mathsf{abc}} + \mathsf{u}^2_{\mathsf{abc}}) \\ P_1, P_2 : \mathsf{y}_3 &= -\lambda^3_{\mathsf{a}}\mathsf{m}_{\mathsf{bc}} - \lambda^3_{\mathsf{b}}\mathsf{m}_{\mathsf{ac}} - \lambda^3_{\mathsf{c}}\mathsf{m}_{\mathsf{ab}} + \gamma^3_{\mathsf{ab}}\mathsf{m}_{\mathsf{c}} \end{aligned}$$

2) P_1 sends y_2 to P_2, while P_2 sends y_1 to P_1, and they locally compute $\mathsf{z} - \mathsf{r} = (\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{y}_3) + \mathsf{m}_{\mathsf{abc}}$.
3) If $\mathsf{isTr} = 1$, P_1, P_2 locally set $\mathsf{p} = (\mathsf{z} - \mathsf{r})^t$, else $\mathsf{p} = \mathsf{z} - \mathsf{r}$. Execute $\Pi_{\mathsf{JSh}}(P_1, P_2, \mathsf{p})$ to generate $[\![\mathsf{p}]\!]$.
4) Locally compute $[\![\mathsf{o}]\!] = [\![\mathsf{p}]\!] + [\![\mathsf{q}]\!]$. Here $\mathsf{o} = \mathsf{z}^t$ if $\mathsf{isTr} = 1$ and z otherwise.
5) *Verification:*
   – Locally compute the following:

$$\begin{aligned} P_3 : \mathsf{v} &= -(\lambda^1_{\mathsf{a}} + \lambda^2_{\mathsf{a}})\mathsf{m}_{\mathsf{bc}} - (\lambda^1_{\mathsf{b}} + \lambda^2_{\mathsf{b}})\mathsf{m}_{\mathsf{ac}} - (\lambda^1_{\mathsf{c}} + \lambda^2_{\mathsf{c}})\mathsf{m}_{\mathsf{ab}} \\ &+ (\gamma^1_{\mathsf{ab}} + \gamma^2_{\mathsf{ab}})\mathsf{m}_{\mathsf{c}} + (\mathsf{w}_{\mathsf{ac}} + \gamma^3_{\mathsf{ac}})\mathsf{m}_{\mathsf{b}} + (\mathsf{w}_{\mathsf{bc}} + \gamma^3_{\mathsf{bc}})\mathsf{m}_{\mathsf{a}} \\ &- (\mathsf{u}^1_{\mathsf{abc}} + \mathsf{u}^2_{\mathsf{abc}} + \mathsf{w}_{\mathsf{abc}}) \\ P_1, P_2 : \mathsf{v}' &= \mathsf{y}_1 + \mathsf{y}_2 + \mathsf{s}_{\mathsf{ac}}\mathsf{m}_{\mathsf{b}} + \mathsf{s}_{\mathsf{bc}}\mathsf{m}_{\mathsf{a}} - \mathsf{s} \end{aligned}$$

   – P_3 sends $\mathsf{H}(\mathsf{v})$ to P_1, P_2, who `abort` iff $\mathsf{H}(\mathsf{v}) \neq \mathsf{H}(\mathsf{v}')$.

[a]For the fair protocol, it is enough for P_0, P_1, P_2 to sample s directly.

Figure 15: 3-input fair multiplication in Tetrad.

4-input multiplication: To obtain $[\![\cdot]\!]$-sharing of $\mathsf{z} = \mathsf{abcd}$ given the $[\![\cdot]\!]$-sharing of $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}$, we can write $\mathsf{z} + \mathsf{r}$ as

$$\begin{aligned} \mathsf{z} - \mathsf{r} &= (\mathsf{m}_{\mathsf{a}} - \lambda_{\mathsf{a}})(\mathsf{m}_{\mathsf{b}} - \lambda_{\mathsf{b}})(\mathsf{m}_{\mathsf{c}} - \lambda_{\mathsf{c}})(\mathsf{m}_{\mathsf{d}} - \lambda_{\mathsf{d}}) - \mathsf{r} \\ &= \mathsf{m}_{\mathsf{abcd}} - \mathsf{m}_{\mathsf{bcd}}\lambda_{\mathsf{a}} - \mathsf{m}_{\mathsf{acd}}\lambda_{\mathsf{b}} - \mathsf{m}_{\mathsf{abd}}\lambda_{\mathsf{c}} - \mathsf{m}_{\mathsf{abc}}\lambda_{\mathsf{d}} \\ &+ \mathsf{m}_{\mathsf{ab}}\gamma_{\mathsf{cd}} + \mathsf{m}_{\mathsf{ac}}\gamma_{\mathsf{bd}} + \mathsf{m}_{\mathsf{ad}}\gamma_{\mathsf{bc}} + \mathsf{m}_{\mathsf{bc}}\gamma_{\mathsf{ad}} + \mathsf{m}_{\mathsf{bd}}\gamma_{\mathsf{ac}} \\ &+ \mathsf{m}_{\mathsf{cd}}\gamma_{\mathsf{ab}} - \mathsf{m}_{\mathsf{a}}\gamma_{\mathsf{bcd}} - \mathsf{m}_{\mathsf{b}}\gamma_{\mathsf{acd}} - \mathsf{m}_{\mathsf{c}}\gamma_{\mathsf{abd}} - \mathsf{m}_{\mathsf{d}}\gamma_{\mathsf{abc}} \\ &+ \gamma_{\mathsf{abcd}} - \mathsf{r} \end{aligned}$$

While the online phase proceeds similarly to the 2 and 3-input multiplication, in the preprocessing phase, the parties need to generate the additive shares of $\gamma_{\mathsf{ab}}, \gamma_{\mathsf{ac}}, \gamma_{\mathsf{ad}}, \gamma_{\mathsf{bc}}, \gamma_{\mathsf{bd}}, \gamma_{\mathsf{cd}}, \gamma_{\mathsf{abc}}, \gamma_{\mathsf{abd}}, \gamma_{\mathsf{acd}}, \gamma_{\mathsf{bcd}}, \gamma_{\mathsf{abcd}}$. This is computed similarly as in the case of 3-input multiplication as follows. Parties generate shares of $\gamma_{\mathsf{ac}}, \gamma_{\mathsf{ad}}, \gamma_{\mathsf{bc}}, \gamma_{\mathsf{bd}}$ similar to the generation of shares of γ_{ac} in the 3-input multiplication. For $\gamma_{\mathsf{ab}}, \gamma_{\mathsf{cd}}$, parties proceed similar to generation of shares of γ_{ab} in the 3-input multiplication, where the respective $\langle\cdot\rangle$-shares are generated. This is followed by generation of shares of $\gamma_{\mathsf{abc}}, \gamma_{\mathsf{abd}}, \gamma_{\mathsf{acd}}, \gamma_{\mathsf{bcd}}, \gamma_{\mathsf{abcd}}$ following steps similar to the ones involved in generating γ_{abc} in the 3-input multiplication. Since the protocol is very similar to the 3-input protocol, we omit the formal details.

c) Bit to Arithmetic: For verifying the $\langle\cdot\rangle$-sharing of u by P_0, we let P_3 obtain the bit $(\lambda_{\mathsf{b}} \oplus r_b)$ as well as its arithmetic equivalent $(\lambda_{\mathsf{b}} \oplus r_b)^{\mathsf{R}}$ in clear. Here r_b denotes a random bit known to P_0, P_1, P_2. P_3 checks if both the received values are equivalent and raise a complaint if they are inconsistent. To catch a corrupt P_0 from sharing a wrong u value, parties use the $\langle\cdot\rangle$-shares of u to compute $(\lambda_{\mathsf{b}} \oplus r_b)^{\mathsf{R}}$. Moreover, the verification steps are designed in such a way that every value communicated can be locally computed by at least two parties. This enables to use jsnd for communication and hence the desired security guarantee is achieved.

Protocol $\Pi_{\mathsf{bit2A}}([\![\mathsf{b}]\!]^{\mathbf{B}})$

Let $\mathsf{u} = (\lambda_{\mathsf{b}})^{\mathsf{R}}$ and $\mathsf{v} = \mathsf{m}^{\mathsf{R}}_{\mathsf{b}}$.

Preprocessing:

1) Generation of $\langle\mathsf{u}\rangle$: P_0, P_3, P_i for $i \in \{1, 2\}$ sample u^i. P_0 sends $\mathsf{u}^3 = \mathsf{u} - \mathsf{u}^1 - \mathsf{u}^2$ to P_1, P_2.
2) P_0, P_1, P_2 sample random $\mathsf{r}_{\mathsf{b}} \in \{0, 1\}$ and $\mathsf{r} \in \mathbb{Z}_{2^\ell}$.
3) P_1, P_2 jsnd $\lambda^3_{\mathsf{b}} \oplus \mathsf{r}_{\mathsf{b}}$ to P_3. P_3 locally sets $\lambda_{\mathsf{b}} \oplus \mathsf{r}_{\mathsf{b}} = (\lambda^1_{\mathsf{b}} \oplus \lambda^2_{\mathsf{b}}) \oplus (\lambda^3_{\mathsf{b}} \oplus \mathsf{r}_{\mathsf{b}})$.
4) Parties compute: $P_1, P_0 : \mathsf{w}_1 = \mathsf{r}^{\mathsf{R}}_{\mathsf{b}} + (\mathsf{u}^1 + \mathsf{u}^3)(1 - 2\mathsf{r}^{\mathsf{R}}_{\mathsf{b}}) + \mathsf{r}$, $\quad P_2, P_0 : \mathsf{w}_2 = (\mathsf{u}^2)(1 - 2\mathsf{r}^{\mathsf{R}}_{\mathsf{b}}) - \mathsf{r}$.
5) P_1, P_0 jsnd w_1 to P_3, while P_2, P_0 jsnd $\mathsf{H}(\mathsf{w}_2)$ to P_3.
6) P_3 sets flag $=$ `continue` if $\mathsf{H}((\lambda_{\mathsf{b}} \oplus r_b)^{\mathsf{R}} - \mathsf{w}_1) = \mathsf{H}(\mathsf{w}_2)$, else flag $=$ `abort`. P_3 sends flag to P_0, P_1, P_2. Parties mutually exchange the flag and accept the value that forms the majority.
7) For robust setting, if flag $=$ `abort`, then $\mathsf{P}_{\mathsf{TP}} = P_1$ (or P_2).

Online: Let $\mathsf{y} = \mathsf{b}^{\mathsf{R}}$.

1) Parties locally compute the following:

$$P_1, P_3 : \mathsf{y}_1 = \mathsf{v} + \mathsf{u}^1(1 - 2\mathsf{v})$$
$$P_2, P_3 : \mathsf{y}_2 = \mathsf{u}^2(1 - 2\mathsf{v})$$
$$P_1, P_2 : \mathsf{y}_3 = \mathsf{u}^3(1 - 2\mathsf{v})$$

2) $(P_1, P_3), (P_2, P_3), (P_1, P_2)$ execute Π_{JSh} on $\mathsf{y}_1, \mathsf{y}_2, \mathsf{y}_3$ to generate the respective $[\![\cdot]\!]$-shares.
3) Compute $[\![\mathsf{y}]\!] = [\![\mathsf{y}_1]\!] + [\![\mathsf{y}_2]\!] + [\![\mathsf{y}_3]\!]$.

Figure 16: Bit to Arithmetic conversion

Lemma C.3 (Communication)**.** *Protocol* Π_{bit2A} *(Fig. 16) requires* $3\ell + 1$ *bits of communication in preprocessing, and* 1 *round and* 3ℓ *bits of communication in the online phase.*

Proof: During preprocessing, generation of $\langle\mathsf{u}\rangle$ involves communication of ℓ bits from P_0 to each of P_1, P_2. As part of verification, two instances of jsnd are executed, one on 1 bit and other on ℓ bits. The communication for hash gets amortized over multiple instances. The online phase involves three instances of joint sharing protocol resulting in 1 rounds and a communication of 3ℓ bits. The costs follow from Lemma B.1. ∎

d) Bit Injection:

Lemma C.4 (Communication). *Protocol Π_{bitInj} requires $6\ell+1$ bits of communication in preprocessing, and 1 round and 3ℓ bits of communication in the online phase.*

Proof: During preprocessing, generation of $\langle \mathsf{u}_i \rangle$ for $i \in [m]$ and its verification is similar to Π_{bit2A}. The cost of generating $\langle \mu_i \rangle$ follows from Π_{MulR}. The communication in the online phase is similar to that of the Π_{bit2A} protocol. The cost follows from Lemma C.3. ∎

e) Piecewise Polynomials:

Lemma C.5 (Communication). *Protocol $\Pi_{\mathsf{piecewise}}$ (Fig. 17) requires $m(6\ell+1)$ bits of communication in preprocessing, and 1 round and 3ℓ bits of communication in the online phase.*

Proof: During preprocessing, generation of $\langle \mathsf{u}_i \rangle, \langle \mu_i \rangle$ for $i \in [m]$ and its verification is similar to Π_{bitInj}. The communication in the online phase is similar to that of the Π_{bitInj} protocol except that parties locally add the values before executing Π_{JSh}. The cost follows from Lemma C.4. ∎

Protocol $\Pi_{\mathsf{piecewise}}\left(\{[\![\mathsf{b}_i]\!]^{\mathbf{B}}, [\![\mathsf{v}_i]\!]\}_{i=1}^{m}\right)$

Let $\mathsf{u}_i = \lambda^{\mathsf{R}}_{\mathsf{b}_i}$ and $\mu_i = \lambda^{\mathsf{R}}_{\mathsf{b}_i}\lambda_{\mathsf{v}_i}$.

Preprocessing: For $i \in [m]$, perform the following:

1) Parties proceed similar to Π_{bit2A} to generate $\langle \mathsf{u}_i \rangle$ (Fig. 16).
2) Generation of $\langle \mu_i \rangle$: Invoke $\Pi_{\mathsf{MulR}}(\mathsf{u}_i, \lambda_{\mathsf{v}_i})$.

Online:

1) Parties locally compute the following:

$$P_1, P_3 : \mathsf{z}_i^1 = \mathsf{m}^{\mathsf{R}}_{\mathsf{b}_i}\mathsf{m}_{\mathsf{v}_i} - \mathsf{m}^{\mathsf{R}}_{\mathsf{b}_i}\lambda^1_{\mathsf{v}_i} + (2\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_i} - 1)(\mu_i^1 - \mathsf{m}_{\mathsf{v}_i}\mathsf{u}_i^1)$$
$$P_2, P_3 : \mathsf{z}_i^2 = -\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_i}\lambda^2_{\mathsf{v}_i} + (2\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_i} - 1)(\mu_i^2 - \mathsf{m}_{\mathsf{v}_i}\mathsf{u}_i^2)$$
$$P_1, P_2 : \mathsf{z}_i^3 = -\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_i}\lambda^3_{\mathsf{v}_i} + (2\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_i} - 1)(\mu_i^3 - \mathsf{m}_{\mathsf{v}_i}\mathsf{u}_i^3)$$

2) Set $\mathsf{z}^1 = \sum_{i=1}^{m} \mathsf{z}_i^1$, $\mathsf{z}^2 = \sum_{i=1}^{m} \mathsf{z}_i^2$, $\mathsf{z}^3 = \sum_{i=1}^{m} \mathsf{z}_i^3$
3) $(P_1, P_3), (P_2, P_3), (P_1, P_2)$ execute Π_{JSh} on $\mathsf{z}^1, \mathsf{z}^2, \mathsf{z}^3$ to generate the respective $[\![\cdot]\!]$-shares.
4) Compute $[\![\mathsf{z}]\!] = [\![\mathsf{z}^1]\!] + [\![\mathsf{z}^2]\!] + [\![\mathsf{z}^3]\!]$.

Figure 17: Piecewise polynomial evaluation protocol

f) Non-Linear Activation functions: We discuss two widely used activation functions, (i) Rectified Linear Unit (ReLU) and (ii) Sigmoid (Sig). These functions can be viewed as piece-wise polynomial functions and can thus be evaluated using the protocol mentioned above ($\Pi_{\mathsf{piecewise}}$, Fig. 17).

(i) ReLU: The ReLU function, $\mathsf{ReLU}(\mathsf{v}) = \max(0, \mathsf{v})$, can be written as a piece-wise polynomial function as follows.

$$\mathsf{ReLU}(\mathsf{v}) = \begin{cases} 0, & \mathsf{v} < 0 \\ \mathsf{v} & 0 \le \mathsf{v} \end{cases}$$

(ii) Sig: We use the MPC-friendly variant of the Sigmoid function [3], [6], [7] which is given below:

$$\mathsf{Sig}(\mathsf{v}) = \begin{cases} 0 & \mathsf{v} < -\frac{1}{2} \\ \mathsf{v} + \frac{1}{2} & -\frac{1}{2} \le \mathsf{v} \le \frac{1}{2} \\ 1 & \frac{1}{2} < \mathsf{v} \end{cases}$$

g) ArgMin/ ArgMax: The formal protocol appears in Fig. 18. Here, $\Pi_{\mathsf{bitext}}([\![\mathsf{x}_1]\!], [\![\mathsf{x}_2]\!])$ computes the boolean sharing corresponding to the msb of $\mathsf{x}_1 - \mathsf{x}_2$.

Protocol $\Pi_{\mathsf{argmin}}([\![\vec{\mathsf{x}}]\!])$

Let $\vec{\mathbf{b}}$ be the bit vector of size m, where m equals the size of $\vec{\mathbf{x}}$. Parties execute the following steps in the respective preprocessing and online phases.

1) If $m = 2$, do the following.
   - $[\![\mathsf{d}_1]\!]^{\mathbf{B}} = \Pi_{\mathsf{bitext}}([\![\mathsf{x}_1]\!], [\![\mathsf{x}_2]\!])$ and $[\![\mathsf{d}_2]\!]^{\mathbf{B}} = 1 \oplus [\![\mathsf{d}_1]\!]^{\mathbf{B}}$.
   - $[\![\mathsf{y}]\!] = \Pi_{\mathsf{obv}}([\![\mathsf{x}_2]\!], [\![\mathsf{x}_1]\!], [\![\mathsf{d}_1]\!]^{\mathbf{B}})$.
   - Return $([\![\mathsf{d}_1]\!]^{\mathbf{B}}, [\![\mathsf{d}_2]\!]^{\mathbf{B}}, [\![\mathsf{y}]\!])$.
2) Else, if $m = 3$, do the following
   - $[\![\mathsf{d}'_1]\!]^{\mathbf{B}} = \Pi_{\mathsf{bitext}}([\![\mathsf{x}_1]\!], [\![\mathsf{x}_2]\!])$.
   - $[\![\mathsf{y}']\!] = \Pi_{\mathsf{obv}}([\![\mathsf{x}_2]\!], [\![\mathsf{x}_1]\!], [\![\mathsf{d}'_1]\!]^{\mathbf{B}})$.
   - $[\![\mathsf{d}'_2]\!]^{\mathbf{B}} = \Pi_{\mathsf{bitext}}([\![\mathsf{y}']\!], [\![\mathsf{x}_3]\!])$.
   - $[\![\mathsf{y}]\!] = \Pi_{\mathsf{obv}}([\![\mathsf{x}_3]\!], [\![\mathsf{y}']\!], [\![\mathsf{d}'_2]\!]^{\mathbf{B}})$.
   - $[\![\mathsf{d}_1]\!]^{\mathbf{B}} = \Pi_{\mathsf{Mult}}([\![\mathsf{d}'_1]\!]^{\mathbf{B}}, [\![\mathsf{d}'_2]\!]^{\mathbf{B}})$, $[\![\mathsf{d}_2]\!]^{\mathbf{B}} = [\![\mathsf{d}'_2]\!]^{\mathbf{B}} \oplus [\![\mathsf{d}_1]\!]^{\mathbf{B}}$.
   - $[\![\mathsf{d}_3]\!]^{\mathbf{B}} = 1 \oplus [\![\mathsf{d}'_1]\!]^{\mathbf{B}} \oplus [\![\mathsf{d}'_2]\!]^{\mathbf{B}}$.
   - Return $([\![\mathsf{d}_1]\!]^{\mathbf{B}}, [\![\mathsf{d}_2]\!]^{\mathbf{B}}, [\![\mathsf{d}_3]\!]^{\mathbf{B}}, [\![\mathsf{y}]\!])$.
3) Else, let $\vec{\mathbf{x_1}} = (\mathsf{x}_1, \ldots, \mathsf{x}_{\lfloor m/2 \rfloor})$ and $\vec{\mathbf{x_2}} = (\mathsf{x}_{\lfloor m/2 \rfloor + 1}, \ldots, \mathsf{x}_m)$.
   - $\left([\![\mathsf{d}_1]\!]^{\mathbf{B}}, \ldots, [\![\mathsf{d}_{\lfloor m/2 \rfloor}]\!]^{\mathbf{B}}, [\![\mathsf{y}_1]\!]\right) = \Pi_{\mathsf{argmin}}([\![\vec{\mathbf{x_1}}]\!])$.
   - $\left([\![\mathsf{d}_{\lfloor m/2 \rfloor+1}]\!]^{\mathbf{B}}, \ldots, [\![\mathsf{d}_m]\!]^{\mathbf{B}}, [\![\mathsf{y}_2]\!]\right) = \Pi_{\mathsf{argmin}}([\![\vec{\mathbf{x_2}}]\!])$.
   - $[\![\mathsf{d}]\!]^{\mathbf{B}} = \Pi_{\mathsf{bitext}}([\![\mathsf{y}_1]\!], [\![\mathsf{y}_2]\!])$.
   - $[\![\mathsf{y}]\!] = \Pi_{\mathsf{obv}}([\![\mathsf{y}_2]\!], [\![\mathsf{y}_1]\!], [\![\mathsf{d}]\!]^{\mathbf{B}})$.
   - $[\![\mathsf{b}_j]\!]^{\mathbf{B}} = \Pi_{\mathsf{Mult}}([\![\mathsf{d}]\!]^{\mathbf{B}}, [\![\mathsf{d}_j]\!]^{\mathbf{B}})$; $j \in \{1, \ldots, \lfloor m/2 \rfloor\}$.
   - $[\![\mathsf{b}_j]\!]^{\mathbf{B}} = \Pi_{\mathsf{Mult}}(1 \oplus [\![\mathsf{d}]\!]^{\mathbf{B}}, [\![\mathsf{d}_j]\!]^{\mathbf{B}})$; $j \in \{\lfloor m/2 \rfloor + 1, \ldots, m\}$.
   - Return $\left([\![\mathsf{b}_1]\!]^{\mathbf{B}}, \ldots, [\![\mathsf{b}_m]\!]^{\mathbf{B}}, [\![\mathsf{y}]\!]\right)$.

Figure 18: Protocol to find index of smallest element in $\vec{\mathbf{x}}$

To begin with, parties initialize $\mathsf{b}_j = 1$ for $\mathsf{b}_j \in \vec{\mathbf{b}}$ by locally setting $\mathsf{m}_{\mathsf{b}_j} = 1$ and $\lambda^1_{\mathsf{b}_j} = \lambda^2_{\mathsf{b}_j} = \lambda^3_{\mathsf{b}_j} = 0$. The minimum, y_{ij}, of two elements, $\mathsf{x}_i, \mathsf{x}_j$ can be computed as: one invocation of bit extraction protocol to obtain $[\![\cdot]\!]^{\mathbf{B}}$-sharing of b_{ij}, where $\mathsf{b}_{ij} = 1$ if $\mathsf{x}_i < \mathsf{x}_j$, and $\mathsf{b}_{ij} = 0$ otherwise;

one invocation of oblivious selection protocol $\Pi_{\mathsf{obv}}(\mathsf{x}_j, \mathsf{x}_i, \mathsf{b}_{ij})$, which outputs $[\![\cdot]\!]$-shares of $\mathsf{y}_{ij} = \mathsf{x}_j$ if $\mathsf{b}_{ij} = 0$, and $\mathsf{y}_{ij} = \mathsf{x}_i$, otherwise. To update $\vec{\mathbf{b}}$ to reflect the pairwise minimums, we view the elements $\mathsf{x}_j \in \vec{\mathbf{x}}$ as the leaves of a binary tree, in a bottom-up manner. For two elements in a pair, say $(\mathsf{x}_i, \mathsf{x}_j)$, whose pairwise minimum is y_{ij}, we let y_{ij} be the root node with x_i as its left child and x_j as its right child. Now, to update $\vec{\mathbf{b}}$, parties multiply b_{ij} with the bits in $\vec{\mathbf{b}}$ associated with the *left-reachable leaf nodes*, which comprise of all the leaf nodes (elements of $\vec{\mathbf{x}}$) that are reachable through the left child of the root. Similarly, parties multiply $1 \oplus \mathsf{b}_{ij}$ with the bits in $\vec{\mathbf{b}}$ associated with the *right-reachable leaf nodes*, which comprise of all the leaf nodes (elements of $\vec{\mathbf{x}}$) that are reachable through the right child of the root. Thus, if $\mathsf{b}_{ij} = 1$ indicating that $\mathsf{x}_i < \mathsf{x}_j$, b_i remains 1 as it gets multiplied by $\mathsf{b}_{ij} = 1$ while b_j gets reset to 0 as it gets multiplied by $1 \oplus \mathsf{b}_{ij} = 0$. The case for $\mathsf{b}_{ij} = 0$ holds for similar reasons. Given the values y_{ij} for the next level, and the updated $\vec{\mathbf{b}}$, the steps are applied recursively until the minimum element is obtained.

The protocol Π_{argmax} which allows the parties to compute the index of the largest element in a $[\![\cdot]\!]$-shared vector $\vec{\mathbf{x}} = (\mathsf{x}_1, \ldots, \mathsf{x}_m)$, is similar to Π_{argmin} with the following difference. To find the maximum among two elements $([\![\mathsf{x}_i]\!], [\![\mathsf{x}_j]\!])$, parties run the bit extraction protocol to obtain $[\![\mathsf{b}_{ij}]\!]^{\mathbf{B}}$ as before, followed by $\Pi_{\mathsf{obv}}(\mathsf{x}_i, \mathsf{x}_j, \mathsf{b}_{ij})$, which outputs $[\![\cdot]\!]$-shares of $\mathsf{y}_{ij} = \mathsf{x}_i$ if $\mathsf{b}_{ij} = 0$, and $\mathsf{y}_{ij} = \mathsf{x}_j$, otherwise. Now, $\vec{\mathbf{b}}$ is updated in each level by multiplying $1 \oplus \mathsf{b}_{ij}$ with the bits in $\vec{\mathbf{b}}$ associated with the *left-reachable leaf nodes* (as described before) and multiplying b_{ij} with the bits in $\vec{\mathbf{b}}$ associated with the *right-reachable leaf nodes*.

APPENDIX D
GARBLED WORLD

A. Garbling scheme and properties

As per Yao's garbling circuit paradigm [11], every wire in the circuit is assigned two κ-bit strings, called "keys", one each for bit value 0 and 1 on that wire. Let $(\mathsf{K}^0_\mathsf{x}, \mathsf{K}^1_\mathsf{x})$ denote the zero-key and one-key, respectively, on wire x in the circuit. For simplicity, the same notation is used for wire identity as well as the value on the wire. For instance, the key-pair for wire x is denoted as $(\mathsf{K}^0_\mathsf{x}, \mathsf{K}^1_\mathsf{x})$, while the key corresponding to bit x on the wire is denoted as $\mathsf{K}^\mathsf{x}_\mathsf{x}$. Then, each gate is constructed by encrypting the output-wire key with the appropriate input-wire keys. For example, for an AND gate with input wires x, y and output wire z, K^0_z is double encrypted with keys $\mathsf{K}^0_\mathsf{x}, \mathsf{K}^0_\mathsf{y}$, with $\mathsf{K}^0_\mathsf{x}, \mathsf{K}^1_\mathsf{y}$, and with $\mathsf{K}^1_\mathsf{x}, \mathsf{K}^0_\mathsf{y}$, while K^1_z is double encrypted with $\mathsf{K}^1_\mathsf{x}, \mathsf{K}^1_\mathsf{y}$. Given one key on each input wire, the output wire key can be obtained by decrypting the ciphertext which was encrypted using the corresponding input wire keys. These ciphertexts are provided in a permuted order so that the evaluating party does not learn which key, K^0_z or K^1_z, it obtains after decryption.

Formally, a garbling scheme $\mathcal{G}$, consists of four algorithms $(\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ defined as follows:

1) $\mathsf{Gb}(1^\kappa, \mathsf{Ckt}) \to (\mathsf{GC}, e, d)$: Gb takes as input the security parameter κ and the circuit Ckt to be garbled, and outputs a garbled circuit GC, encoding information e and decoding information d.
2) $\mathsf{En}(x, e) \to \mathbf{X}$: En encodes input x using e to output encoded input $\mathbf{X}$. $\mathbf{X}$ is referred to as encoded input or encoded keys interchangeably.
3) $\mathsf{Ev}(\mathsf{GC}, \mathbf{X}) \to \mathbf{Y}$: Ev evaluates the garbled circuit GC on the encoded input $\mathbf{X}$ and produces the encoded output $\mathbf{Y}$.
4) $\mathsf{De}(\mathbf{Y}, d) \to y$: The encoded output $\mathbf{Y}$ is decoded into the clear output y by running the De algorithm on $\mathbf{Y}$ and d.

We rely on the following properties of garbling scheme [59] in our constructions.

1) A garbling scheme $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ is *correct* if for all input lengths $n \leq \mathsf{poly}(\kappa)$, circuits $C : \{0,1\}^n \to \{0,1\}^m$ and inputs $x \in \{0,1\}^n$, the following holds.
$$\Pr[\mathsf{De}(\mathsf{Ev}(\mathsf{GC}, \mathsf{En}(x, e)), d) \neq C(x) : (\mathsf{GC}, e, d) \leftarrow \mathsf{Gb}(1^\kappa, C)] < \mathsf{negl}(\kappa)$$
2) A garbling scheme $\mathcal{G}$ is said to be *private* if for all $n \leq \mathsf{poly}(\kappa)$, circuit $C : \{0,1\}^n \to \{0,1\}^m$, there exists a PPT simulator $\mathcal{S}_{\mathbf{priv}}$ such that for all $x \in \{0,1\}^n$, for all PPT adversary $\mathcal{A}$ the following distributions are computationally indistinguishable.
- REAL(C, x): run $(\mathsf{GC}, e, d) \leftarrow \mathsf{Gb}(1^\kappa, C)$ and output $(\mathsf{GC}, \mathsf{En}(x, e), d)$.
- IDEAL$(C, C(x))$: run $(\mathsf{GC}', \mathbf{X}, d') \leftarrow \mathcal{S}_{\mathbf{priv}}(1^\kappa, C, C(x))$ and output $(\mathsf{GC}', \mathbf{X}, d')$.
3) A garbling scheme $\mathcal{G}$ is *authentic* if for all $n \leq \mathsf{poly}(\kappa)$, circuit $C : \{0,1\}^n \to \{0,1\}^m$, input $x \in \{0,1\}^n$ and for all PPT adversary $\mathcal{A}$, the following probability is $\mathsf{negl}(\kappa)$.
$$\Pr\begin{pmatrix} \hat{\mathbf{Y}} \neq \mathsf{Ev}(\mathsf{GC}, \mathbf{X}) & \mathbf{X} = \mathsf{En}(x, e), (\mathsf{GC}, e, d) \leftarrow \mathsf{Gb}(\kappa, \mathsf{Ckt}), \\ \wedge\, \mathsf{De}(\hat{\mathbf{Y}}, d) \neq \bot & : \hat{\mathbf{Y}} \leftarrow \mathcal{A}(\mathsf{GC}, \mathbf{X}) \end{pmatrix}$$

B. 2GC Variant

We begin with the details of the evaluation and output phases.

a) Evaluation: Let $f(\mathsf{x})$ be the function to be evaluated. At this point, the function input is $[\![\cdot]\!]^{\mathbf{C}}$-shared. This renders $[\![\cdot]\!]^{\mathbf{G}}$-sharing for the input of the GC that corresponds to the function $f'(\mathsf{m}_\mathsf{x}, \alpha_\mathsf{x}, \lambda^3_\mathsf{x})$ which first combines the given boolean-shares to compute the actual input and then applies f on it. Let GC_j denote the garbled circuit to be sent to $P_j \in \{P_1, P_2\}$ by garblers in Φ_j. Sending of GC_j is overlapped with the key transfer (during generation of $[\![\mathsf{x}]\!]^{\mathbf{C}}$), to save rounds, where garblers in $\{P_0, P_3\}$ jsnd GC_j to P_j. On receiving the GC, evaluators evaluate their respective GCs and obtain the key corresponding to the output, say z. This generates $[\![\mathsf{z}]\!]^{\mathbf{G}}$.

b) Output phase: The goal of output computation is to compute the output z from $[\![\mathsf{z}]\!]^{\mathbf{G}}$. To reconstruct z towards $P_j \in \{P_1, P_2\}$, two garblers in Φ_j send the least significant bit p^j of $\mathsf{K}^{0,j}_\mathsf{z}$, referred to as the decoding information, to P_j. If the received values are consistent, P_j uses the received p^j to reconstruct z as $\mathsf{z} = \mathsf{p}^j \oplus \mathsf{q}^j$, where q^j denotes the least significant bit of $\mathsf{K}^{\mathsf{z},j}_\mathsf{z}$; else P_j aborts. To reconstruct z towards the garblers $P_g \in \{P_0, P_3\}$, one evaluator, say P_1 sends the least significant bit, q^1, of $\mathsf{K}^{\mathsf{z},1}_\mathsf{z}$ along with $\mathcal{H} = \mathsf{H}(\mathsf{K}^{\mathsf{z},1}_\mathsf{z})$ to

P_g, where H is a collision-resistant hash function. If a garbler received a consistent $(\mathsf{q}^1, \mathcal{H})$ pair from P_1 such that there exists a $K \in \{\mathsf{K}_z^{0,1}, \mathsf{K}_z^{1,1}\}$ whose least significant bit is q^1 and $\mathsf{H}(K) = \mathcal{H}$, then it uses q^1 for reconstructing z; else the garbler aborts the computation. Note that a corrupt evaluator P_1 cannot create confusion among garblers in $\{P_0, P_3\}$ by sending the key that was not output by the GC owing to the authenticity of the garbling scheme. Reconstruction is lightweight and requires a single round for garblers while reconstruction towards evaluators can be overlapped with key transfer and does not incur extra rounds. The protocol appears in Fig. 19.

Protocol $\Pi^{\mathbf{G}}_{\mathsf{Rec}}(\mathcal{P}, [\![\mathsf{z}]\!]^{\mathbf{G}})$

1) For an output wire z, let p^j denote the least significant bit of $\mathsf{K}_z^{0,j}$ and q^j denote the least significant bit of $\mathsf{K}_z^{z,j}$ for $j \in \{1,2\}$.
2) *Reconstruction towards* $P_j \in \{P_1, P_2\}$: Garblers P_0, P_3 in Φ_j jsnd p^j to P_j. If P_j received consistent values from P_0, P_3, it reconstructs z as $\mathsf{z} = \mathsf{p}^j \oplus \mathsf{q}^j$.
3) *Reconstruction towards* $P_g \in \{P_0, P_3\}$: P_1 sends q^1 and $\mathcal{H} = \mathsf{H}(\mathsf{K}_z^{z,1})$ to P_g, where H is a collision-resistant hash function. P_g uses the q^1 received from P_1 for reconstructing z as $\mathsf{z} = \mathsf{p}^1 \oplus \mathsf{q}^1$ if there exists a $K \in \{\mathsf{K}_z^{0,1}, \mathsf{K}_z^{1,1}\}$ whose least significant bit is q^1 and $\mathsf{H}(K) = \mathcal{H}$.

Figure 19: Output computation: reconstruction of z

c) Optimizations when deployed in mixed framework: Working in the preprocessing model enables transfer of the (communication-intensive) GC and generating $[\![\cdot]\!]^{\mathbf{G}}$-shares of the input-independent shares of x (i.e. α_x, λ_x^3) in the preprocessing phase. Thus, the online phase is very light and only requires one round to generate $[\![\cdot]\!]^{\mathbf{G}}$-shares for the input-dependent data (i.e. m_x). Since evaluation is local, evaluators obtain $[\![\cdot]\!]^{\mathbf{G}}$-sharing of the GC output at the end of 1 round.

d) Achieving fairness and robustness: To ensure fairness, we require a fair reconstruction protocol which proceeds as follows. As described in §III-C, parties first ensure that all parties are alive. If so, they proceed similar to the protocol in Fig. 19, except with the following differences. For reconstruction towards evaluators, *all* three respective garblers send it the decoding information. The evaluator selects the value appearing in majority for reconstruction. For reconstruction towards garblers P_0, P_3, *both* the evaluators send the least significant bit of the output key together with its hash to the garbler. The presence of at least one honest evaluator guarantees that both garblers will be on the same page.

To achieve robustness, the main difference from its fair counterpart is use of a robust jsnd primitive. This guarantees that in the event that a misbehaviour is detected, a $\mathsf{P_{TP}}$ is identified which can take the computation to completion and deliver the output to all.

C. 1 GC Variant

The input $\mathsf{x} = \mathsf{x}_1 \oplus \mathsf{x}_2 \oplus \mathsf{x}_3$ for this variant consists of the shares, $\mathsf{x}_1 = \mathsf{m}_x \oplus \lambda_x^2$ and $\mathsf{x}_2 = \lambda_x^3, \mathsf{x}_3 = \lambda_x^1$, where $\mathsf{m}_x, \lambda_x^1, \lambda_x^2, \lambda_x^3$ are as defined in $[\![\mathsf{x}]\!]^{\mathbf{B}}$. While keys for the GC are sampled by all three garblers P_0, P_2, P_3, it suffices for only P_0, P_3 to generate and jsnd the GC to evaluator P_1, and P_2 assists only in the key transfer. Elaborately, the common input x_3 held by P_0, P_3 is hard-coded in the circuit before being garbled by them. This necessitates a key transfer only for inputs x_1 and x_2. Garblers P_0, P_2, P_3 generate keys for the inputs following a similar procedure as in the 2GC variant. Then, P_2, P_3 jsnd the key for x_1 to P_1 while garblers P_0, P_2 jsnd the key for x_2.

The evaluation and output phases are similar to the 2GC variant except that now there exists only a single garbling instance. Looking ahead, in the mixed protocol framework, the output has to be reconstructed towards P_1, P_2. Reconstruction towards P_1 does not incur additional rounds since sending of decoding information can be overlapped with key transfer. However, unlike in the 2GC variant where reconstruction towards P_2 can be done similar to reconstruction towards P_1, in the 1GC variant an additional round is required as P_2 is no longer an evaluator. This incurs one extra round as opposed to the 2GC variant.

a) Achieving fairness: To ensure fair reconstruction (§III-C), parties first perform an aliveness check. Following this, they proceed towards fair reconstruction of z from $[\![\mathsf{z}]\!]^{\mathbf{G}}$ as follows. First, reconstruction of z is carried out towards the garblers $P_g \in \Phi_1$. For this, P_1 sends q (least significant bit of K_z^{z}) and $\mathcal{H} = \mathsf{H}(\mathsf{K}_z^{z})$ to P_g as before. Now, if a garbler received a consistent $(\mathsf{q}, \mathcal{H})$ pair from P_1 such that there exists a $K \in \{\mathsf{K}_z^0, \mathsf{K}_z^1\}$ whose least significant bit is q and $\mathsf{H}(K) = \mathcal{H}$, then it uses q for reconstructing z, and sends z to its co-garblers. Else, a garbler accepts a z received from a co-garbler as the output. Thus, further dissemination of the output by garblers ensures that all parties are on the same page. If garblers receive the output, reconstruction of z is carried out towards P_1. For this, all garblers (who received the output) send the decoding information to P_1 who selects the majority value to reconstruct z.

b) Achieving robustness: To attain robustness, we list below the differences from the fair protocol that have to be carried out. The first difference is use of a robust variant of jsnd. Second, in input sharing protocol, where x_1 is held by only garbler P_0, a corrupt P_0 may refrain from providing P_1 with the correct key (sent as the opening information for the commitment). To ensure robustness, in the event that P_1 fails to receive the correct key from P_0, we let P_1 complain to all parties about this inconsistency by sending an inconsistency bit. All parties exchange this inconsistency bit among themselves, and agree on the majority value. If all parties agree on the presence of an inconsistency, then P_0, P_1 are identified to be in conflict and $\mathsf{P_{TP}} = P_2$ is set to carry out the rest of the computation. Finally, to ensure a robust reconstruction, the following approach is taken. Observe that the fair reconstruction provides robustness as long as evaluator P_1 is honest. In the event when none of the garblers obtain the output in the fair protocol, it is guaranteed that evaluator P_1 is corrupt. Thus, in such a scenario, all parties take P_1 to be corrupt, and proceed with P_0 as the $\mathsf{P_{TP}}$.

Appendix E
Mixed Framework

a) Arithmetic to Boolean Conversion: The protocol for arithmetic to boolean conversion appears in Fig. 20.

Protocol Π_{A2B}

Preprocessing: P_0, P_3 execute joint boolean sharing to generate $[\![\mathsf{v}_2]\!]^{\mathbf{B}}$, where $\mathsf{v}_2 = -(\lambda_{\mathsf{v}}^1 + \lambda_{\mathsf{v}}^2)$.

Online:

1) P_1, P_2 execute joint boolean sharing to generate $[\![\mathsf{v}_1]\!]^{\mathbf{B}}$, where $\mathsf{v}_1 = \mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}^3$.
2) Parties obtain $[\![\mathsf{v}]\!]^{\mathbf{B}} = [\![\mathsf{v}_1]\!]^{\mathbf{B}} + [\![\mathsf{v}_2]\!]^{\mathbf{B}}$ using addition circuit.

Figure 20: Arithmetic to Boolean Conversion

b) Boolean to Arithmetic Conversion: The protocol for arithmetic to boolean conversion appears in Fig. 21. We remark that the protocol Π_{B2A} can be used to efficiently generate edaBits [27] in our setting. For this, the parties non-interactively generate the boolean sharing for ℓ-bits and perform the Π_{B2A} conversion to obtain the equivalent arithmetic value.

Protocol $\Pi_{\mathsf{B2A}}(\mathcal{P}, [\![\mathsf{v}]\!]^{\mathbf{B}})$

Let v_i denote the ith bit of v. Let $\lambda_{\mathsf{v}i} = \lambda_{\mathsf{v}_i}^1 \oplus \lambda_{\mathsf{v}_i}^2 \oplus \lambda_{\mathsf{v}_i}^3$, $\mathsf{p}_i = (\mathsf{m}_{\mathsf{v}i})^{\mathsf{R}}$, and $\mathsf{q} = (\lambda_{\mathsf{v}i})^{\mathsf{R}}$

Preprocessing:

1) For $i \in \{0, 1, \ldots, \ell-1\}$, parties execute the preprocessing of Π_{bit2A} (Fig. 16) for each bit v_i of v, to generate $\langle \mathsf{q}_i \rangle = (\mathsf{q}_i^1, \mathsf{q}_i^2, \mathsf{q}_i^3)$.

Online: Let $\mathsf{y}_i = \mathsf{v}_i^{\mathsf{R}}$ and y denotes the arithmetic equivalent of v.

1) Parties locally compute the following:

$$P_1, P_3 : \mathsf{y}^1 = \sum_{i=0}^{\ell-1} 2^i \mathsf{y}_i^1 = \sum_{i=0}^{\ell-1} 2^i(\mathsf{p}_i + \mathsf{q}_i^1(1 - 2\mathsf{p}_i))$$

$$P_2, P_3 : \mathsf{y}^2 = \sum_{i=0}^{\ell-1} 2^i \mathsf{y}_i^2 = \sum_{i=0}^{\ell-1} 2^i(\mathsf{q}_i^2(1 - 2\mathsf{p}_i))$$

$$P_1, P_2 : \mathsf{y}^3 = \sum_{i=0}^{\ell-1} 2^i \mathsf{y}_i^3 = \sum_{i=0}^{\ell-1} 2^i(\mathsf{q}_i^3(1 - 2\mathsf{p}_i))$$

2) $(P_1, P_3), (P_2, P_3), (P_1, P_2)$ execute Π_{JSh} on $\mathsf{y}^1, \mathsf{y}^2, \mathsf{y}^3$ to generate the respective $[\![\cdot]\!]$-shares.
3) Parties locally compute $[\![\mathsf{y}]\!] = [\![\mathsf{y}^1]\!] + [\![\mathsf{y}^2]\!] + [\![\mathsf{y}^3]\!]$.

Figure 21: Boolean to Arithmetic Conversion

c) End-to-end Conversions: Table XI, Table XII compare our sharing conversions with Trident [4]. The cost for the 2GC variant of Trident is computed by incorporating a parallel execution, where P_3 is additionally made an evaluator together with P_0. For uniformity, we consider a function, F, to be computed on an ℓ-bit inputs x, y using a garbled circuit (GC) in the mixed framework, which gives an ℓ-bit output $\mathsf{z} = \mathsf{F}(\mathsf{x}, \mathsf{y})$, where ℓ denotes the ring size in bits. Let G^{F} denote the corresponding GC. In the table, G^{S2} denotes a 2-input garbled subtraction circuit; $\hat{\mathsf{G}}$ denotes the garbled circuit with decoding information; $\mathsf{G}^{\mathsf{n_1} \times 1, \ldots, \mathsf{n_m} \times \mathsf{m}}$ denotes $\mathsf{n_i}$ instances of GC G^{i} for $i \in \{1, \ldots, \mathsf{m}\}$ and $|\mathsf{G}^{\mathsf{n_1} \times 1, \ldots, \mathsf{n_m} \times \mathsf{m}}|$ denotes the collective size.

Protocol[a]	Reference	Communication[b] (Preprocessing)	Rounds (Online)	Communication (Online)
A-G-A	Trident Tetrad	$2\lvert\hat{\mathsf{G}}^{2\times\mathsf{S2},\mathsf{F}}\rvert + 6\ell\kappa + \ell$	2 1	$4\ell\kappa + 2\ell$ $4\ell\kappa + \ell$
A-G-B	Trident Tetrad	$2\lvert\mathsf{G}^{\mathsf{S2},\mathsf{F}}\rvert + 6\ell\kappa + \ell$	2 1	$4\ell\kappa + 2\ell$ $4\ell\kappa + \ell$
B-G-A	Trident Tetrad	$2\lvert\hat{\mathsf{G}}^{\mathsf{S2},\mathsf{F}}\rvert + 6\ell\kappa + \ell$	2 1	$4\ell\kappa + 2\ell$ $4\ell\kappa + \ell$
B-G-B	Trident Tetrad	$2\lvert\mathsf{G}^{\mathsf{F}}\rvert + 6\ell\kappa + \ell$	2 1	$4\ell\kappa + 2\ell$ $4\ell\kappa + \ell$

[a] A: arithmetic, B: boolean, G: garbled
[b] Notations: ℓ - size of ring in bits, κ - computational security parameter.

Table XI: Conversions (2GC variant): Trident [4] and Tetrad.

Protocol[a]	Reference	Communication[b] (Preprocessing)	Rounds (Online)	Communication (Online)
A-G-A	Trident Tetrad	$\lvert\hat{\mathsf{G}}^{2\times\mathsf{S2},\mathsf{F}}\rvert + 3\ell\kappa + \ell$	2 2	$2\ell\kappa + 3\ell$ $2\ell\kappa + 2\ell$
A-G-B	Trident Tetrad	$\lvert\mathsf{G}^{\mathsf{S2},\mathsf{F}}\rvert + 3\ell\kappa + \ell$	2 2	$2\ell\kappa + 3\ell$ $2\ell\kappa + 2\ell$
B-G-A	Trident Tetrad	$\lvert\hat{\mathsf{G}}^{\mathsf{S2},\mathsf{F}}\rvert + 3\ell\kappa + \ell$	2 2	$2\ell\kappa + 3\ell$ $2\ell\kappa + 2\ell$
B-G-B	Trident Tetrad	$\lvert\mathsf{G}^{\mathsf{F}}\rvert + 3\ell\kappa + \ell$	2 2	$2\ell\kappa + 3\ell$ $2\ell\kappa + 2\ell$

[a] A: arithmetic, B: boolean, G: garbled
[b] Notations: ℓ - size of ring in bits, κ - computational security parameter.

Table XII: Conversions (1GC variant): Trident [4] and Tetrad.

Appendix F
ML Algorithms

a) Training and Inference of NN: An NN can be divided into various layers, where each layer contains a pre-defined number of nodes. These nodes are a linear function composed of a non-linear "activation" function. The nodes at the input layer are evaluated on the input features to evaluate a neural network. The outputs from these nodes are fed as inputs to the nodes in the next layer. This process is repeated for all the layers to obtain the output. The underlying operation involved is a computation of activation matrices for all the layers. This constitutes the forward propagation phase. The backward propagation involves adjusting model parameters according to the difference in the computed output and the actual output and comprises computing error matrices.

Concretely, each layer comprises matrix multiplications followed by an application of the ReLU function. The maxpool layer additionally follows convolutional layers after the ReLU layer. After evaluating the layers in a sequential manner, at the output layer, we use the MPC friendly variant of the softmax activation function, $\mathsf{softmax}(u_i) = \frac{\mathsf{ReLU}(u_i)}{\sum_{j=1}^{\mathsf{n}} \mathsf{ReLU}(u_j)}$, proposed by SecureML [7]. To perform the division, we switch from arithmetic to garbled world and then use a division garbled circuit [60] followed by a switch back to the arithmetic world. For training, we use Gradient Descent, where the forward propagation comprises computing activation matrices for all the layers in the network. The backward propagation comprises computing error matrices involving matrix multiplications with derivative of maxpool and derivative of ReLU, depending on the network architecture. We refer readers to [4], [6]–[8], [51] for formal details.

b) Inference of SVM: SVM is a function which takes as input an n-dimensional *feature vector*, $\vec{\mathbf{x}}$, and outputs the *category* to which the feature vector belongs. SVM is implemented as a matrix $\mathbf{F}$, of dimension $q \times n$ where each row of $\mathbf{F}$ is called the support vector and a vector $\vec{\mathbf{b}} = (b_1, \ldots, b_q)$, is called the *bias*. Each element of $\mathbf{F}$ and $\vec{\mathbf{b}}$ lies in $\mathbb{Z}_{2^\ell}$. Each support vector along with a scalar from the bias can classify the input $\vec{\mathbf{x}}$ into a specific category. More precisely, let $\mathbf{F}_i$ denote the i^{th} row of matrix $\mathbf{F}$. Then, the value $\mathbf{F}_i \cdot \vec{\mathbf{x}} + b_i$ specifies how likely $\vec{\mathbf{x}}$ is to be in category i. To find the most likely category, we compute argmax over these values, i.e. $\text{category}(\vec{\mathbf{x}}) = \text{argmax}_{i \in \{1,\ldots,q\}} \mathbf{F}_i \cdot \vec{\mathbf{x}} + b_i$. We refer the readers to [18] for more details.

APPENDIX G
SECURITY PROOFS

Without loss of generality, we prove the security of our robust framework. The case for fairness follows similarly, and we omit its details. We provide proofs in the $\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{jsnd}}$-hybrid model, where $\mathcal{F}_{\mathsf{setup}}$ (Fig. 9), $\mathcal{F}_{\mathsf{jsnd}}$ (Fig. 23) denote the ideal functionality for the shared-key setup and jsnd, respectively.

The strategy for simulating the computation of function f (represented by a circuit Ckt) is as follows: Simulation begins with the simulator emulating the shared-key setup ($\mathcal{F}_{\mathsf{setup}}$) functionality and giving the respective keys to the adversary. This is followed by the input sharing phase in which $\mathcal{S}$ computes the input of $\mathcal{A}$, using the known keys, and sets the inputs of the honest parties, to be used in the simulation, to 0. $\mathcal{S}$ invokes the ideal functionality $\mathcal{F}_{\text{ROBUST}}$ on behalf of $\mathcal{A}$ using the extracted input and obtains the output y. $\mathcal{S}$ now knows the inputs of $\mathcal{A}$ and can compute all the intermediate values for each of the building blocks. $\mathcal{S}$ proceeds with simulating each of the building blocks in the topological order.

For modularity, we provide the simulation steps for each building block (arithmetic/garbled) separately. Carrying out these blocks in the topological order yields the simulation for the entire computation. If a $\mathsf{P_{TP}}$ is identified during the simulation, the simulator stops and returns the function output to the adversary on behalf of the $\mathsf{P_{TP}}$ as per $\mathcal{F}_{\mathsf{jsnd}}$.

a) Ideal jsnd *Functionality:* The ideal jsnd functionality for fairness security appears in Fig. 22 and that for the robust setting appears in Fig. 23.

Functionality $\mathcal{F}_{\mathsf{jsnd}}$ (for fair security)

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

Step 1: $\mathcal{F}_{\mathsf{jsnd}}$ receives $(\mathsf{Input}, \mathsf{v}_s)$ from senders P_s for $s \in \{i, j\}$, $(\mathsf{Input}, \perp)$ from receiver P_k and fourth party P_l. While sending the inputs, the adversary is also allowed to send a special `abort` command.

Step 2: Set $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_l = \perp$.

Step 3: If $\mathsf{v}_i = \mathsf{v}_j$, set $\mathsf{msg}_k = \mathsf{v}_i$. Else, set $\mathsf{msg}_k = \mathtt{abort}$.

Step 4: Send $(\mathsf{Output}, \mathsf{msg}_s)$ to P_s for $s \in \{0, 1, 2, 3\}$.

Figure 22: Ideal functionality for jsnd in Tetrad

Functionality $\mathcal{F}_{\mathsf{jsnd}}$ (for robust security)

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

Step 1: $\mathcal{F}_{\mathsf{jsnd}}$ receives $(\mathsf{Input}, \mathsf{v}_s)$ from senders P_s for $s \in \{i, j\}$, $(\mathsf{Input}, \perp)$ from receiver P_k and fourth party P_l, while it receives $(\mathsf{select}, \mathsf{ttp})$ from $\mathcal{S}$. Here ttp is a boolean value, with a 1 indicating that $\mathsf{P_{TP}} = P_l$ should be established.

Step 2: If $\mathsf{v}_i = \mathsf{v}_j$ and $\mathsf{ttp} = 0$, or if $\mathcal{S}$ has corrupted P_l[a], set $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_l = \perp, \mathsf{msg}_k = \mathsf{v}_i$ and go to **Step 4**.

Step 3: Else, set $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_k = \mathsf{msg}_l = P_l$.

Step 4: Send $(\mathsf{Output}, \mathsf{msg}_s)$ to P_s for $s \in \{0, 1, 2, 3\}$.

[a] This condition is used to capture the fact that a corrupt P_l cannot create an inconsistency in $\mathcal{F}_{\mathsf{jsnd}}$ since the parties actively involved in $\mathcal{F}_{\mathsf{jsnd}}$ would be honest

Figure 23: Ideal functionality for robust jsnd [14]

A. Arithmetic/Boolean World

We provide the simulation for the case for corrupt P_0, P_1 and P_3. The case for corrupt P_2 is similar to that of P_1.

a) Sharing Protocol (Π_{Sh}, Fig. 1): During the preprocessing, $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$ emulates $\mathcal{F}_{\mathsf{setup}}$ and gives the respective keys to $\mathcal{A}$. The values commonly held with $\mathcal{A}$ are sampled using the respective keys, while others are sampled randomly. The details for the online phase are provided next. We omit the simulation for corrupt P_3 as it is similar to that of P_1, P_2.

Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$

Online:

– If dealer is $\mathcal{A}$, $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$ receives $\mathsf{m_v}$ from $\mathcal{A}$ on behalf of P_1, P_2, P_3. If the received values are consistent, $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$ computes $\mathcal{A}$'s input v as $\mathsf{v} = \mathsf{m_v} - [\lambda_\mathsf{v}]_1 - [\lambda_\mathsf{v}]_2 - [\lambda_\mathsf{v}]_3$, else sets v as the default value. It invokes $\mathcal{F}_{\text{ROBUST}}$ on $(\mathsf{Input}, \mathsf{v})$ to obtain the function output y.

– If dealer is P_1, P_2 or P_3, there is nothing to simulate as P_0 doesn't receive any value during the protocol.

Figure 24: Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$ for corrupt P_0

Simulator $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Sh}}}$

Online:

– If dealer is $\mathcal{A}$, $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Sh}}}$ receives $\mathsf{m_v}$ from $\mathcal{A}$ on behalf of P_2, P_3. If the received values are consistent, $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Sh}}}$ computes $\mathcal{A}$'s input v as $\mathsf{v} = \mathsf{m_v} - [\lambda_\mathsf{v}]_1 - [\lambda_\mathsf{v}]_2 - [\lambda_\mathsf{v}]_3$, else sets v as the default value. It invokes $\mathcal{F}_{\text{ROBUST}}$ on $(\mathsf{Input}, \mathsf{v})$ to obtain the function output y.

– If dealer is P_0, P_2 or P_3, $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Sh}}}$ sets $\mathsf{v} = 0$ and performs the protocol steps honestly.

Figure 25: Simulator $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Sh}}}$ for corrupt P_1

Shares unknown to $\mathcal{A}$ are sampled randomly in the simulation, whereas in the real protocol, they are sampled using the pseudorandom function (PRF). The indistinguishability of the simulation thus follows by a reduction to the security of the PRF. The same holds for the rest of the blocks.

The simulation for the joint sharing protocol (Π_{JSh}) is similar to that of the sharing protocol. The protocol's design is such that the simulator will always know the value to be sent as part of the joint sharing protocol. The communication is constituted by jsnd calls and is emulated according to the simulation of $\mathcal{F}_{\mathsf{jsnd}}$.

b) Multiplication Protocol (Π_{Mult}):

Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Mult}}}$

Preprocessing:

– Computes $\gamma^1_{\mathsf{ab}}, \gamma^2_{\mathsf{ab}}$, and γ^3_{ab} on behalf of P_1, P_2, P_3.

– Samples $\mathsf{u}^1, \mathsf{u}^2$ using the respective keys with $\mathcal{A}$ and computes r. The joint sharing of q is simulated as discussed earlier.

– Receives w from $\mathcal{A}$ on behalf of P_3.

– Simulating Π_{VrfyP0}: Joint sharing of $\mathsf{e}_1, \mathsf{e}_2, \mathsf{e}$ is simulated as discussed earlier. The rest of the steps are simulated honestly. This is possible since $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Mult}}}$ knows the randomness and inputs that should be used by $\mathcal{A}$.

Online: P_0 has no communication in the online phase except the jsnd instances which are emulated by $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Mult}}}$.

Figure 26: Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Mult}}}$ for corrupt P_0

Simulator $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Mult}}}$

Preprocessing:

– Computes $\gamma^1_{\mathsf{ab}}, \gamma^2_{\mathsf{ab}}$, and γ^3_{ab} on behalf of P_0, P_2, P_3.

– Samples u^1 using the respective keys with $\mathcal{A}$. Samples a random u^2 and computes r. The joint sharing of q is simulated as discussed earlier.

– Simulate the steps of Π_{VrfyP0} honestly.

Online:

– Computes $\mathsf{y}_1 + \mathsf{s}_1, \mathsf{y}_2 + \mathsf{s}_2, \mathsf{y}_3$ honestly.

– Emulates two instances of $\mathcal{F}_{\mathsf{jsnd}}$ – i) $\mathcal{A}$ as sender to send $\mathsf{y}_1 + \mathsf{s}_1$ to P_2, and ii) $\mathcal{A}$ as receiver to obtain $\mathsf{y}_2 + \mathsf{s}_2$ from P_2.

– Simulates joint sharing as discussed earlier.

Figure 27: Simulator $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Mult}}}$ for corrupt P_1

Simulator $\mathcal{S}^{P_3}_{\Pi_{\mathsf{Mult}}}$

Preprocessing:

– Computes $\gamma^1_{\mathsf{ab}}, \gamma^2_{\mathsf{ab}}$, and γ^3_{ab} on behalf of P_0, P_1, P_2.

– Samples $\mathsf{u}^1, \mathsf{u}^2$ using the respective keys with $\mathcal{A}$ and computes r. The joint sharing of q is simulated as discussed earlier.

– Honestly computes and sends w to $\mathcal{A}$.

– Simulate the steps of Π_{VrfyP0} honestly.

Online:

– Computes $\mathsf{y}_1 + \mathsf{s}_1, \mathsf{y}_2 + \mathsf{s}_2, \mathsf{y}_3$ honestly.

– Emulates two instances of $\mathcal{F}_{\mathsf{jsnd}}$ with $\mathcal{A}$ as sender to exchange $\mathsf{y}_1 + \mathsf{s}_1, \mathsf{y}_2 + \mathsf{s}_2$ among P_1, P_2.

– Simulates joint sharing as discussed earlier.

Figure 28: Simulator $\mathcal{S}^{P_3}_{\Pi_{\mathsf{Mult}}}$ for corrupt P_3

c) Reconstruction Protocol (Π_{Rec}, Fig. 13): Using the input of $\mathcal{A}$ obtained during simulation of sharing protocol, $\mathcal{S}_{\Pi_{\mathsf{Rec}}}$ invokes $\mathcal{F}_{\text{ROBUST}}$ on behalf of $\mathcal{A}$ and obtains the function output y in clear. $\mathcal{S}_{\Pi_{\mathsf{Rec}}}$ calculates the missing share of $\mathcal{A}$ using y and the other shares. The missing share is then communicated to $\mathcal{A}$ by emulating the $\mathcal{F}_{\mathsf{jsnd}}$ functionality.

B. Security Proof for Garbled World

In this section, we present the proof of security for our robust GC protocol with 2GCs. The case for 1 GC is similar, and we omit the details. For completeness, we provide the simulation assuming function evaluation entirely through the GC. However, as in the previous section, simulation steps are provided for the different phases separately. Thus, the simulation for the appropriate phase can be used while simulating the entire protocol in the mixed framework.

The simulation begins with the simulator emulating the shared-key setup ($\mathcal{F}_{\mathsf{setup}}$) functionality and giving the respective keys to the adversary. This is followed by the input sharing phase in which $\mathcal{S}$ computes the input of $\mathcal{A}$, using the known keys, and sets the inputs of the honest parties, to be used in the simulation, to 0. $\mathcal{S}$ invokes the ideal functionality $\mathcal{F}_{\text{ROBUST}}$ on behalf of $\mathcal{A}$ using the extracted input and obtains the output y. $\mathcal{S}$ proceeds with simulating the GC computation phase using the output y by invoking the privacy simulator for the GC. The reconstruction phase follows this. We provide the simulation steps in the following order:

- Generation of boolean shares for the input.
- Transfer of keys and GC to the evaluator.
- Output computation.

We give the proof with respect to a corrupt P_0 and a corrupt P_1. Proofs for corrupt P_3 and corrupt P_2 follow similar to proof for corrupt P_0 and P_1, respectively.

a) Generation of boolean shares for the input: This simulation proceeds as per the simulation of the boolean world mentioned in §G-A.

b) Key, GC transfer and evaluation: The simulation for $\Pi^{\mathbf{G}}_{\mathsf{Sh}}$ coupled with the GC transfer for a corrupt P_1 and corrupt P_0 are provided here. Cases for corrupt P_2, P_3 follow.

Simulator $\mathcal{S}^{P_0}_{\mathsf{Ev}}$

– With respect to the j^{th} garbling instance for $j \in \{1, 2\}$, $\mathcal{S}^{P_0}_{\mathsf{Ev}}$ generates the keys $\{\mathsf{K}^{\mathsf{b},j}_{\mathsf{m_x}}, \mathsf{K}^{\mathsf{b},j}_{\alpha_{\mathsf{x}}}, \mathsf{K}^{\mathsf{b},j}_{\lambda^3_{\mathsf{x}}}\}_{\mathsf{b} \in \{0,1\}}$ for each function input x and the GC as per the honest execution.

– Sends the keys for $\mathsf{K}^{\mathsf{m_x},j}_{\mathsf{m_x}}, \mathsf{K}^{\alpha_{\mathsf{x}},j}_{\alpha_{\mathsf{x}}}$ and GC_j to P_j for $j \in \{1, 2\}$ by emulating $\mathcal{F}_{\mathsf{jsnd}}$ with $\mathcal{A}$ as the sender.

Figure 29: Simulator $\mathcal{S}^{P_0}_{\mathsf{Ev}}$ for corrupt P_0

Simulator $\mathcal{S}^{P_1}_{\mathsf{Ev}}$

– With respect to the first garbling instance, $\mathcal{S}^{P_1}_{\mathsf{Ev}}$ runs $(\mathsf{GC}_1, \mathbf{X}_1, d_1) \leftarrow \mathcal{S}_{\mathbf{priv}}(1^\kappa, \mathsf{Ckt}, \mathsf{y})$ where y is obtained via invoking $\mathcal{F}_{\text{ROBUST}}$ on $\mathcal{A}$'s input. With respect to the second garbling instance, $\mathcal{S}^{P_1}_{\mathsf{Ev}}$ generates the keys $\{\mathsf{K}^{\mathsf{b},2}_{\mathsf{m_x}}, \mathsf{K}^{\mathsf{b},2}_{\alpha_{\mathsf{x}}}, \mathsf{K}^{\mathsf{b},2}_{\lambda^3_{\mathsf{x}}}\}_{\mathsf{b} \in \{0,1\}}$ for each function input x and GC_2 as per the honest execution.

– $\mathcal{S}_{\mathsf{Ev}}^{P_1}$ sends the keys for each input v to the GC, and GC_1 by emulating $\mathcal{F}_{\mathsf{jsnd}}$ with $\mathcal{A}$ as the receiver.

– $\mathcal{S}_{\mathsf{Ev}}^{P_1}$ emulates $\mathcal{F}_{\mathsf{jsnd}}$ together with $\mathcal{A}$ as the sender to send $\mathsf{K}_{\mathsf{m_x}}^{\mathsf{m_x},2}, \mathsf{K}_{\lambda_{\mathsf{x}}^3}^{\lambda_{\mathsf{x}}^3,2}$ to P_2.

Figure 30: Simulator $\mathcal{S}_{\mathsf{Ev}}^{P_1}$ for corrupt P_1

c) Output computation:

Simulator $\mathcal{S}_{\mathsf{Rec}}^{P_0}$

– Let $\mathsf{lsb}(\mathsf{v})$ denote the least significant bit of v.

– $\mathcal{S}_{\mathsf{Rec}}^{P_0}$ sends $\mathsf{q}^J = \mathsf{y} \oplus \mathsf{lsb}(\mathsf{K}_{\mathsf{y}}^{0,j})$ and $\mathcal{H}^j = \mathsf{H}(\mathsf{K})$ to $\mathcal{A}$ on behalf of honest $P_j \in \mathcal{E}$ such that $\mathsf{K} \in \{\mathsf{K}_{\mathsf{y}}^{0,j}, \mathsf{K}_{\mathsf{y}}^{1,j}\}$ and $\mathsf{q}^j = \mathsf{lsb}(\mathsf{K})$, where y is obtained via invoking $\mathcal{F}_{\text{ROBUST}}$.

Figure 31: Simulator $\mathcal{S}_{\mathsf{Rec}}^{P_0}$ for corrupt P_0

Simulator $\mathcal{S}_{\mathsf{Rec}}^{P_1}$

– Let $\mathsf{lsb}(\mathsf{v})$ denote the least significant bit of v.

– $\mathcal{S}_{\mathsf{Rec}}^{P_1}$ sends $\mathsf{p}^1 = \mathsf{lsb}(\mathsf{K}_{\mathsf{y}}^{0,1})$ to $\mathcal{A}$ on behalf of honest garblers in Φ_1 where y is obtained via invoking $\mathcal{F}_{\text{ROBUST}}$.

Figure 32: Simulator $\mathcal{S}_{\mathsf{Rec}}^{P_1}$ for corrupt P_1

d) Indistinguishability argument: We argue that $\text{IDEAL}_{\mathcal{F},\mathcal{S}_\Pi} \overset{c}{\approx} \text{REAL}_{\Pi,\mathcal{A}}$ when $\mathcal{A}$ corrupts P_1 based on the following series of intermediate hybrids.

HYB_0: Same as $\text{REAL}_{\Pi,\mathcal{A}}$.

HYB_1: Same as HYB_0, except that P_0, P_2, P_3 use uniform randomness instead of pseudo-randomness to sample values not known to P_1.

HYB_2: Same as HYB_1 except that GC_1 is created as $(\mathsf{GC}_1, \mathbf{X}_1, d_1) \leftarrow \mathcal{S}_{\mathsf{prv}}(1^\kappa, \mathsf{Ckt}, \mathsf{y})$.

Since $\text{HYB}_2 := \text{IDEAL}_{\mathcal{F},\mathcal{S}_\Pi}$, to conclude the proof we show that every two consecutive hybrids are indistinguishable.

$\text{HYB}_0 \overset{c}{\approx} \text{HYB}_1$: The difference between the hybrids is that P_0, P_2, P_3 use uniform randomness in HYB_1 rather than pseudo-randomness as in HYB_0 (for sampling $[\alpha]_2$). The indistinguishability follows via reduction to the security of the PRF.

$\text{HYB}_1 \overset{c}{\approx} \text{HYB}_2$: The difference between the hybrids is in the way $(\mathsf{GC}_1, \mathbf{X}_1, d_1)$ is generated. In HYB_1, $(\mathsf{GC}_1, e_1, d_1) \leftarrow \mathsf{Gb}(1^\kappa, \mathsf{Ckt})$ is run. In HYB_2, it is generated as $(\mathsf{GC}_1, \mathbf{X}_1, d_1) \leftarrow \mathcal{S}_{\mathsf{prv}}(1^\kappa, \mathsf{Ckt}, \mathsf{y})$. Indistinguishability follows via reduction to the privacy of the garbling scheme.

We argue that $\text{IDEAL}_{\mathcal{F},\mathcal{S}_\Pi} \overset{c}{\approx} \text{REAL}_{\Pi,\mathcal{A}}$ when $\mathcal{A}$ corrupts P_0 based on the following series of intermediate hybrids.

HYB_0: Same as $\text{REAL}_{\Pi,\mathcal{A}}$.

HYB_1: Same as HYB_0, except that P_1, P_2, P_3 use uniform randomness instead of pseudo-randomness to sample values not known to P_0.

HYB_2: Same as HYB_1 except that hash of the key K where $\mathsf{K} \in \{\mathsf{K}_{\mathsf{y}}^{0,j}, \mathsf{K}_{\mathsf{y}}^{1,j}\}$ to be sent to $\mathcal{A}$ is computed such that $\mathsf{lsb}(\mathsf{K}) \oplus \mathsf{lsb}(\mathsf{K}_{\mathsf{y}}^{0,j}) = \mathsf{y}$, for $j \in \{1,2\}$ instead of obtaining it as output of GC evaluation.

Since $\text{HYB}_2 := \text{IDEAL}_{\mathcal{F},\mathcal{S}_\Pi}$, to conclude the proof we show that every two consecutive hybrids are indistinguishable.

$\text{HYB}_0 \overset{c}{\approx} \text{HYB}_1$: The difference between the hybrids is that P_1, P_2, P_3 use uniform randomness in HYB_1 rather than pseudo-randomness as in HYB_0 (for sampling $\lambda 3$). The indistinguishability follows via reduction to the security of the PRF.

$\text{HYB}_1 \overset{c}{\approx} \text{HYB}_2$: The difference between the hybrids is that in HYB_1, key K where $\mathsf{K} \in \{\mathsf{K}_{\mathsf{y}}^{0,j}, \mathsf{K}_{\mathsf{y}}^{1,j}\}$ for $j \in \{1,2\}$ is computed as output of the GC evaluation while in HYB_2, it is computed such that $\mathsf{lsb}(\mathsf{K}) \oplus \mathsf{lsb}(\mathsf{K}_{\mathsf{y}}^{0,j}) = \mathsf{y}$. Due to the correctness of the garbling scheme, the equivalence of K computed in both the hybrids holds.