Tetrad: Actively Secure 4PC for Secure Training and Inference

Nishat Koti
Indian Institute of Science

Arpita Patra
Indian Institute of Science

Rahul Rachuri
Aarhus University

Ajith Suresh
Indian Institute of Science

Abstract

In this work, we design an efficient mixed-protocol framework, *Tetrad*, with applications to privacy-preserving machine learning. It is designed for the four-party setting with at most one active corruption and supports rings.

Our fair multiplication protocol requires communicating only 5 ring elements improving over the state-of-the-art protocol of Trident (Chaudhari et al. NDSS'20). The technical highlights of Tetrad include efficient (a) truncation without any overhead, (b) multi-input multiplication protocols for arithmetic and boolean worlds, (c) garbled-world, tailor-made for the mixed-protocol framework, and (d) conversion mechanisms to switch between the computation styles. The fair framework is also extended to provide robustness without inflating the costs.

The competence of Tetrad is tested with benchmarks for deep neural networks such as LeNet and VGG16, and support vector machines. One variant of our framework aims at minimizing the execution time, while the other focuses on the monetary cost. We observe improvements up to 6× over Trident across these parameters.

1 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 the privacy of the underlying data. PPML can be deployed across various domains such as healthcare, recommendation systems, text translation, etc., with works like [4] 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 [11, 14, 15, 38, 41, 43, 46, 49, 55] have been gaining interest, where a server enacts the role of a party in the MPC protocol. MPC [25, 57] 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 [11, 15, 26, 33]. 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 with feature-size independence:* 4PC offers a simpler and 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 simple in terms of design (as shown in this and prior works). To enhance practical efficiency, many recent works [15, 19, 30, 46] 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 [18–20] propose MPC protocols over 32 or 64 bit rings to leverage CPU optimizations.

MPC protocols can be categorized as high-throughput [2, 5, 6, 14, 15, 23, 33, 41, 45, 46] and low-latency [12, 13], where the former, based on secret-sharing, requires less communication

# Parties	Work	#Active Parties	Security	Dot Product		Dot Product with Truncation			Conversions
				Comm$_{pre}$	Comm$_{on}$	Comm$_{pre}$	Comm$_{on}$	Rounds$_{on}$	
3	ABY3 [41]	3	Abort	$12d\ell$	$9d\ell$	$12d\ell+84\ell$	$9d\ell+3\ell$	2	A-B-G
	BLAZE [46]	2	Fair	3ℓ	3ℓ	15ℓ	3ℓ	1	A-B
	SWIFT (3PC) [33]	2	GOD	3ℓ	3ℓ	15ℓ	3ℓ	1	A-B
4	Mazloom et al. [39]	4	Abort	2ℓ	4ℓ	2ℓ	4ℓ	1	A-B
	Trident [15]	3	Fair	3ℓ	3ℓ	6ℓ	3ℓ	1	A-B-G
	Tetrad	2	Fair	2ℓ	3ℓ	2ℓ	3ℓ	1	A-B-G
	SWIFT (4PC) [33]	2	GOD	3ℓ	3ℓ	4ℓ	3ℓ	1	A-B
	Fantastic Four [17]	3	GOD	-	$6(\ell+\kappa)$	$76(\ell+\kappa)+54x+12$	$9\ell+6\kappa$	>1	A-B
	Tetrad-R$^{\mathsf{I}}$	2	GOD	2ℓ	3ℓ	2ℓ	3ℓ	1	A-B-G
	Tetrad-R$^{\mathsf{II}}$	2	GOD	3ℓ	3ℓ	3ℓ	3ℓ	1	A-B-G

ℓ - size of ring in bits, κ - security parameter, d - length of the vectors, x - number of bits for the fractional part in FPA semantics.
'Comm' - communication, 'pre' - preprocessing, 'on' - online; A, B, G indicate support for arithmetic, boolean, and garbled worlds respectively.

Table 1: Comparison of actively-secure MPC frameworks (3PC and 4PC) for PPML

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 motivated mixed protocols where each protocol is executed in a world where it performs best. Mixed-protocol frameworks [15, 20, 21, 41, 43, 45, 49, 51] 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 in an honest majority setting with active security.

Works such as [39, 41, 55] 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 resources-expensive tasks such as PPML, as causing an abort will waste its resources. Trident [15] showed that the stronger notion of fairness can be achieved at the cost of abort. In cases where the risk of failure for 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 next.

1.1 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, we may want different levels of security. For this, we propose multiple variants of the framework, covering fairness (**Tetrad**) and robustness (**Tetrad-R$^{\mathsf{I}}$**, **Tetrad-R$^{\mathsf{II}}$**) guarantees. This fair variant improves upon the state-of-the-art *fair* framework of Trident [15]. Our robust frameworks offer support for secure training, which was not supported in previous works such as [33].

1.1.1 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 [15].

Robust multiplication in Tetrad-R$^{\mathsf{I}}$, retains the same (amortized) communication cost as that of the fair protocol but uses a verification check in the preprocessing over extended rings. In fact, for large circuits ($\sim 2^{20}$ multiplications), the overhead amortizes, making Tetrad-R$^{\mathsf{I}}$ as efficient as its fair counterpart. In other words, for large circuits, robustness comes for free over fairness. On the other hand, multiplication in Tetrad-R$^{\mathsf{II}}$ does away with the computation over extended rings. It requires a minimal overhead of 1 element communication in the preprocessing for multiplication over Tetrad.

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

On-demand applications. The design allows us 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. These variants take 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.

Truncation without any overhead. Multiplication (and dot product) with truncation forms an essential component to retain the FPA semantics while performing PPML operations. Inspired by [39] which provides protocols satisfying security with abort, we demonstrate for the *first time*, in the fair and robust settings, how multiplication (and dot-product) with truncation can be performed without any extra cost.

Multi-input multiplication. Inspired by [44, 45], we propose new protocols for 3 and 4-input multiplication, allowing multiplication of 3 and 4 inputs in one shot. 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, our multi-input multiplication, when coupled with the optimized parallel prefix adder circuit from [45], brings in a 2× improvement in online rounds. It also cuts down the online communication of secure comparison, factoring into improvements in PPML applications.

1.1.2 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. They use the garbled world 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 – 1) Tailor-made GC-based protocols and 2) *end-to-end* conversion techniques.

1) Garbled world: The tailor-made GC-based (fair and robust) protocols, when deployed in the mixed framework, offer the following impactful features – i) amortized round complexity of 1, ii) no use of commitments for the inputs as opposed to the work of [13, 29], and iii) no requirement of an explicit input sharing and output reconstruction phase [13], as the garbled protocol only forms an intermediate part of the complete computation. The construction requires 2 GC communication with just one online round. However, for applications where communication is a bottleneck, we demonstrate how the protocol can be realized with 1 GC communication at the expense of one additional online round.

2) End-to-end Conversions: Departing from existing methods we provide for the first time, *end-to-end* conversion techniques such as Arithmetic-Garbled-Arithmetic. The standard approach until now was to perform a *piece-wise* combination of *Arithmetic to Garbled* followed by a *Garbled to Arithmetic* conversion. End-to-end conversions benefit from not having to generate a full-fledged garbled-shared output after the computation. Instead, these conversions aim to produce a "partial" garbled-shared output that is enough to lead to an arithmetic sharing of the output. This results in end-to-end conversions of the form "x-Garbled-x" where x can be either arithmetic or boolean that need just a single round for our garbled world (cf. Table 8) as opposed to the two in Trident [15].

Comparison of Tetrad with actively secure PPML frameworks in 3PC and 4PC is presented in Table 1. The dot product is chosen as a parameter as it is one of the most crucial building blocks in PPML applications.

1.1.3 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 [35] and VGG16 [53] and the inference phase of Support Vector Machines are benchmarked.

The implementation section is presented through the lens of deployment scenarios with 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 the different scenarios.

	Training & Inference			Training	Inference
Ref	Time_{on}	Com_{tot}	CT_{tot}	Cost*	TP_{on}
Tetrad_{T}, Tetrad-R_{T}	●	◐	●	◐	●
Tetrad_{C}, Tetrad-R_{C}	◐	●	◐	●	◐
Trident	○	○	○	○	○

- 'Com' - Communication, 'Time' - Runtime, 'CT' - Cumulative Runtime, 'Cost' - Monetary Cost, 'TP_{on}' - Throughput, on - online, tot - total.
- ○ - good, ◐ - better, ● - best, (w.r.t parameter considered).
- Cost of Trident is lower than Tetrad-R_{T}.

Table 2: Comparison of Trident [15] with the versions of Tetrad for deep neural networks (cf. NN-4 in §6).

Considering online runtime as the metric, Tetrad_{T}, Tetrad-R_{T} are the time-optimized (T) variants, with the fastest online phase of all. Tetrad_{C}, Tetrad-R_{C} are the cost-optimized (C) variants, minimizing deployment cost. This is measured via *monetary cost* [47], which helps to capture the effect of the combined total runtime of the parties, and communication. All the variants are compared against Trident [15], and their relative performance is indicated in Table 2. The comparison is made over four main factors – run time, communication, monetary cost (cf. Table 4), and throughput.

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 2 shows that Tetrad is better compared to Trident across all the parameters considered. Within Tetrad, $\mathsf{Tetrad_T}$ fare better when it comes to online run time for both training and inference, while $\mathsf{Tetrad_C}$ do better in terms of communication. When it comes to inference, throughput is more relevant than the cost, and here, the time-optimized variants fare the best. Robust variants follow the same trends, and the reasons behind them are elaborated in §6.

2 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, the 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 [22, 52, 54]. 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) [11, 14, 15, 41, 46] 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) [11, 14, 15, 41, 46] 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 2.1. *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 2.2. *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 $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$.

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) [15], where a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is split into four shares. 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 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 $\mathsf{m_v}$, and $\mathsf{m_v} = \mathsf{v} + \lambda_{\mathsf{v}}$.

Type	P_0	P_1	P_2	P_3
$[\cdot]$-sharing	–	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	$(\lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^2, \lambda_{\mathsf{v}}^3)$	$(\mathsf{m_v}, \lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^3)$	$(\mathsf{m_v}, \lambda_{\mathsf{v}}^2, \lambda_{\mathsf{v}}^3)$	$(\mathsf{m_v}, \lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^2)$

$\mathsf{v} = \mathsf{v}_1 + \mathsf{v}_2\ (+\mathsf{v}_3)$ and $\mathsf{m_v} = \mathsf{v} + \lambda_{\mathsf{v}}$

Table 3: 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, and so it is only active towards the end of the computation. Complying with the roles and RSS format, the distribution is done as follows: $P_0 : \{\lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^2, \lambda_{\mathsf{v}}^3\}, P_1 : \{\lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^3, \mathsf{m_v}\}, P_2 : \{\lambda_{\mathsf{v}}^2, \lambda_{\mathsf{v}}^3, \mathsf{m_v}\}$, and $P_3 : \{\lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^2, \mathsf{m_v}\}$. The shares are distributed among $\mathcal{D}$ such that P_3 gets $\mathsf{m_v}$ whereas P_0 gets all the shares of λ_{v}. In the preprocessing phase, P_0 computes a part of the data needed for verification (cf. Fig. 1) 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. §6). The RSS sharing semantics is presented in Table 3, denoted by $[\![\cdot]\!]$, in a modular way with the help of three intermediate sharing semantics $[\cdot], (\!(\cdot)\!)$ and $\langle\cdot\rangle$. All the sharings used are linear i.e. given sharings 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 non-interactively for an integer m.

Notation 2.3. *(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* **B**, *and* **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 §4 with the relevant context.

3 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 §3.1. The multiplication protocol with abort is presented in §3.2, followed by the details on elevating the security to fairness in §3.2.1. Lastly, in §3.2.2, we show how to improve the security to robustness[1]. Formal details along with cost analysis for the protocols has been deferred to §B.

3.1 Primitives

Joint-Send (jsnd)**.** The Joint-Send (jsnd) primitive allows to 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 `abort` 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.

Joint-Send (jsnd) **for robust protocols.** To achieve robustness, we instantiate jsnd using the joint-message passing (jmp) primitive of [33]. The jsnd primitive (Fig. 9) allows two senders P_i, P_j to relay a common message, $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, to recipient P_k, either by ensuring successful delivery of v, or by establishing a Trusted Third Party (TTP) among the parties. The instantiation of jmp 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 TTP identification). This requires 1 round of interaction and ℓ bits of communication. To leverage amortization, *verify* will be executed only once, at the end the computation, requiring 2 rounds.

Note that the appropriate instantiation of jsnd is used depending on the security guarantee. For simplicity, protocols where the fair and robust variants only differ in the instantiation of jsnd used, we give a common construction for both.

[1] The classical notion of robustness is achieved

Notation 3.1. *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 $\Pi_{\mathsf{jsnd}}(P_i, P_j, \mathsf{v}, P_k)$.*

Sharing. Protocol Π_{Sh} (Fig. 10) 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.2) in a way that P_i will get the entire mask λ_{v}. During the online phase, P_i computes $\mathsf{m}_{\mathsf{v}} = \mathsf{v} + \lambda_{\mathsf{v}}$ 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.

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 $\mathsf{m}_{\mathsf{v}} = \mathsf{v} + \lambda_{\mathsf{v}}$ 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 $\lambda_{\mathsf{v}}^1 \in_R \mathbb{Z}_{2^\ell}$; Parties set $\lambda_{\mathsf{v}}^2 = \mathsf{m}_{\mathsf{v}} = 0$; 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 $\lambda_{\mathsf{v}}^3 \in_R \mathbb{Z}_{2^\ell}$; Parties set $\lambda_{\mathsf{v}}^1 = \mathsf{m}_{\mathsf{v}} = 0$; 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 $\lambda_{\mathsf{v}}^2 \in_R \mathbb{Z}_{2^\ell}$; Parties set $\lambda_{\mathsf{v}}^3 = \mathsf{m}_{\mathsf{v}} = 0$; P_0, P_3 jsnd $\lambda_{\mathsf{v}}^1 = -\mathsf{v} - \lambda_{\mathsf{v}}^1$ to P_1.

Reconstruction. Protocol $\Pi_{\mathsf{Rec}}(\mathcal{P}, \mathsf{v})$ (Fig. 11) 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.

3.2 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.

Online. Note that,

$$\begin{aligned} \mathsf{z} - \mathsf{r} &= \mathsf{ab} - \mathsf{r} = (\mathsf{m_a} - \lambda_{\mathsf{a}})(\mathsf{m_b} - \lambda_{\mathsf{b}}) - \mathsf{r} \\ &= \mathsf{m_{ab}} - \mathsf{m_a}\lambda_{\mathsf{b}} - \mathsf{m_b}\lambda_{\mathsf{a}} + \gamma_{\mathsf{ab}} - \mathsf{r} \quad \text{(cf. notation 2.3)} \end{aligned} \qquad (1)$$

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 as 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 P_1, P_2.

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

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

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. Parties locally compute the following:

$$\begin{aligned} P_0, P_1 &: \gamma_{\mathsf{ab}}^1 = \lambda_{\mathsf{a}}^1 \lambda_{\mathsf{b}}^3 + \lambda_{\mathsf{a}}^3 \lambda_{\mathsf{b}}^1 + \lambda_{\mathsf{a}}^3 \lambda_{\mathsf{b}}^3 \\ P_0, P_2 &: \gamma_{\mathsf{ab}}^2 = \lambda_{\mathsf{a}}^2 \lambda_{\mathsf{b}}^3 + \lambda_{\mathsf{a}}^3 \lambda_{\mathsf{b}}^2 + \lambda_{\mathsf{a}}^2 \lambda_{\mathsf{b}}^2 \\ P_0, P_3 &: \gamma_{\mathsf{ab}}^3 = \lambda_{\mathsf{a}}^1 \lambda_{\mathsf{b}}^2 + \lambda_{\mathsf{a}}^2 \lambda_{\mathsf{b}}^1 + \lambda_{\mathsf{a}}^1 \lambda_{\mathsf{b}}^1 \end{aligned}$$

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 $\mathsf{u}^1 + \mathsf{u}^2 = \gamma_{\mathsf{ab}}^3 - \mathsf{r}$ for a random $\mathsf{r} \in_R \mathbb{Z}_{2^\ell}$.
3. P_0, P_3 compute $\mathsf{r} = \gamma_{\mathsf{ab}}^3 - \mathsf{u}^1 - \mathsf{u}^2$ 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 $\mathsf{w} = \gamma_{\mathsf{ab}}^1 + \gamma_{\mathsf{ab}}^2 + \mathsf{s}$ to P_3.

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

1. Parties locally compute the following:

$$\begin{aligned} P_1 &: \mathsf{y}_1 = -\lambda_{\mathsf{a}}^1 \mathsf{m}_{\mathsf{b}} - \lambda_{\mathsf{b}}^1 \mathsf{m}_{\mathsf{a}} + \gamma_{\mathsf{ab}}^1 + \mathsf{u}^1 \\ P_2 &: \mathsf{y}_2 = -\lambda_{\mathsf{a}}^2 \mathsf{m}_{\mathsf{b}} - \lambda_{\mathsf{b}}^2 \mathsf{m}_{\mathsf{a}} + \gamma_{\mathsf{ab}}^2 + \mathsf{u}^2 \\ P_1, P_2 &: \mathsf{y}_3 = -\lambda_{\mathsf{a}}^3 \mathsf{m}_{\mathsf{b}} - \lambda_{\mathsf{b}}^3 \mathsf{m}_{\mathsf{a}} \end{aligned}$$

2. P_1 sends y_1 to P_2, while P_2 sends y_2 to P_1, and they locally compute $\mathsf{z} - \mathsf{r} = (\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{y}_3) + \mathsf{m}_{\mathsf{a}} \mathsf{m}_{\mathsf{b}}$.
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. 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 $\mathsf{v} = -(\lambda_{\mathsf{a}}^1 + \lambda_{\mathsf{a}}^2) \mathsf{m}_{\mathsf{b}} - (\lambda_{\mathsf{b}}^1 + \lambda_{\mathsf{b}}^2) \mathsf{m}_{\mathsf{a}} + \mathsf{u}^1 + \mathsf{u}^2 + \mathsf{w}$ 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 1: Multiplication with / without truncation in Tetrad.

Verification. To ensure the correctness of the values exchanged, 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.

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 $\gamma_{\mathsf{ab}} = \gamma_{\mathsf{ab}}^1 + \gamma_{\mathsf{ab}}^2 + \gamma_{\mathsf{ab}}^3$, where P_0 along with P_i can compute γ_{ab}^i 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 $\gamma_{\mathsf{ab}}^i + \left[\gamma_{\mathsf{ab}}^3 - r\right]$. Instead of sampling a random r, P_0, P_3, along with P_i, sample the share for $\gamma_{\mathsf{ab}}^3 - \mathsf{r}$ as u^i for $i \in \{1,2\}$. P_0, P_3 compute r as $\gamma_{\mathsf{ab}}^3 - \mathsf{u}^1 - \mathsf{u}^2$.

For iii), P_3 needs $\mathsf{w} = \gamma_{\mathsf{ab}}^1 + \gamma_{\mathsf{ab}}^2 + \mathsf{s}$. 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 $\gamma_{\mathsf{ab}}^i + \mathsf{s}_i$ 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.

Truncation. For a value $\mathsf{v} = \mathsf{v}_1 + \mathsf{v}_2$, SecureML [43] showed that the truncated value $\mathsf{v}/2^x$, denoted by v^t, is equivalent to $\mathsf{v}_1^t + \mathsf{v}_2^t$, with very high probability. The design of our multiplication allows for truncation to be carried out this way without any additional overhead in communication. Observe that $\mathsf{z}^t = (\mathsf{z} - \mathsf{r})^t + \mathsf{r}^t$. 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.

Multiplication by a constant 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 $\beta^1 = \alpha.(\mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}^3)$ and $\beta^2 = \alpha.(-\lambda_{\mathsf{v}}^1 - \lambda_{\mathsf{v}}^2)$. 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.

3.2.1 Achieving Fairness

Here, we show how to extend the security of Tetrad from abort to fairness using techniques from Trident [15]. 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.

3.2.2 Achieving Robustness

In this section, we show how to extend the security of Tetrad to robustness. We provide two variants with different trade-offs in the communication for multiplication. i) Tetrad-R$^{\mathsf{I}}$: It has the same amortised communication complexity as that of Tetrad but requires verification in the preprocessing phase over Galois rings. ii) Tetrad-R$^{\mathsf{II}}$: It avoids operating over Galois ring (and operates entirely over $\mathbb{Z}_{2^\ell}$) as that in Tetrad-R$^{\mathsf{I}}$ but incurs a communication overhead of 1 element in the preprocessing phase over Tetrad.

Tetrad-R$^{\mathsf{I}}$. On a high level, we make two modifications to the multiplication protocol Π_{Mult} (Fig. 1). In the preprocessing, communication comes from a Π_{JSh} in step 3 of the protocol, and the value w sent by P_0 to P_3, in step 4. To get robustness, the robust variant of Π_{JSh} is used. To ensure the correctness of w, we introduce Π_{VrfyP0} (Fig. 2). If Π_{VrfyP0} fails, parties identify a TTP in the preprocessing phase itself. The second modification is in the online phase, which proceeds as that of Π_{Mult}. If any `abort` happens, P_0 is assigned as the TTP. Since P_0 does not participate in the online phase of the 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. 1) protocol, P_0 computes and sends $\mathsf{w} = \gamma^1_{\mathsf{ab}} + \gamma^2_{\mathsf{ab}} + \mathsf{s}_1 + \mathsf{s}_2$ to P_3 with P_0, P_1, P_2 knowing $\mathsf{s}_1, \mathsf{s}_2$ 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 reduces to verifying $\mathsf{w} = \mathsf{w}^1 + \mathsf{w}^2$.

To verify this relation for all M multiplication gates in the circuit, i.e. $\{\mathsf{w}_j \stackrel{?}{=} \mathsf{w}^1_j + \mathsf{w}^2_j\}_{j\in[M]}$, a naive solution (that works over fields) is to compute a random linear combination and verify the relation on the sum. In detail, parties sample M random values, $\tau_1, \ldots, \tau_M$ and compute the following: $P_0, P_1 : \mathsf{e}^1 = \sum_{j=1}^M \tau_j \mathsf{w}^1_j$; $P_0, P_2 : \mathsf{e}^2 = \sum_{j=1}^M \tau_j \mathsf{w}^2_j$; $P_0, P_3 : \mathsf{e} = \sum_{j=1}^M \tau_j \mathsf{w}_j$. Each of these pairs of parties can generate the respective $[\![\cdot]\!]$-sharing by executing Π_{JSh}. Then they invoke a robust reconstruction on $[\![\mathsf{e} - \mathsf{e}^1 - \mathsf{e}^2]\!]$ and check if it is 0. If not, one among P_1, P_2, P_3 is assigned as a TTP. However, this solution will not work over rings as not every element in the ring has an inverse, as opposed to in fields. Hence we perform the check over a Galois ring [1, 10].

To carry out the verification, the extended ring $\mathbb{Z}_{2^\ell}/f(x)$ is used, which is the ring of all polynomials with coefficients in $\mathbb{Z}_{2^\ell}$ modulo an irreducible polynomial f of degree d over $\mathbb{Z}_2$. Here, each element in $\mathbb{Z}_{2^\ell}$ is lifted to a d-degree polynomial in $\mathbb{Z}_{2^\ell}[x]/f(x)$ (which results in blowing up the communication by a factor d). Given this, to verify the M values, further packing is performed. More concretely, assume that d divides M and $M = d \cdot q$. For $j = 1, \ldots, q$, public polynomial g_j and shared polynomials g^1_j and g^2_j are defined for each set of d values $\{\mathsf{w}, \mathsf{w}^1, \mathsf{w}^2\}$, all of which are then combined to check whether $\{\mathsf{w}_j \stackrel{?}{=} \mathsf{w}^1_j + \mathsf{w}^2_j\}_{j\in[M]}$. We describe the polynomial with respect to $j = 1$ below.

$$g_1 = \mathsf{w}_1 + X \cdot \mathsf{w}_2 + \ldots + X^{d-1} \cdot \mathsf{w}_d$$
$$g^1_1 = \mathsf{w}^1_1 + X \cdot \mathsf{w}^1_2 + \ldots + X^{d-1} \cdot \mathsf{w}^1_d$$
$$g^2_1 = \mathsf{w}^2_1 + X \cdot \mathsf{w}^2_2 + \ldots + X^{d-1} \cdot \mathsf{w}^2_d$$

Now, parties sample random values $\mathsf{r}_1, \ldots, \mathsf{r}_q \in \mathbb{Z}_{2^\ell}/f(x)$ and compute $g = \sum_{j=1}^q \mathsf{r}_j g_j$, $g^1 = \sum_{j=1}^q \mathsf{r}_j g^1_j$ and $g^2 = \sum_{j=1}^q \mathsf{r}_j g^2_j$. This is followed by robustly reconstructing $g - g^1 - g^2$ and verifying if this value is 0. If not, P_0 is identified to be a corrupt and computation is carried out by a TTP. The formal verification protocol appears in Fig. 2.

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

1. Define the following polynomials over $\mathbb{Z}_{2^\ell}/f(x)$ for $j \in [q]$.

$$g_j = \mathsf{w}_{1+(j-1)d} + X \cdot \mathsf{w}_{2+(j-1)d} + \ldots + X^{d-1} \cdot \mathsf{w}_{d+(j-1)d}$$
$$g^1_j = \mathsf{w}^1_{1+(j-1)d} + X \cdot \mathsf{w}^1_{2+(j-1)d} + \ldots + X^{d-1} \cdot \mathsf{w}^1_{d+(j-1)d}$$
$$g^2_j = \mathsf{w}^2_{1+(j-1)d} + X \cdot \mathsf{w}^2_{2+(j-1)d} + \ldots + X^{d-1} \cdot \mathsf{w}^2_{d+(j-1)d}$$

2. Parties generate random values $\mathsf{r}_1, \ldots, \mathsf{r}_q \in \mathbb{Z}_{2^\ell}/f(x)$, and compute $g = \sum_{j=1}^q \mathsf{r}_j g_j$, $g^1 = \sum_{j=1}^q \mathsf{r}_j g^1_j$ and $g^2 = \sum_{j=1}^q \mathsf{r}_j g^2_j$.
3. Parties execute $\Pi_{\mathsf{JSh}}(P_0, P_1, g^1)$, $\Pi_{\mathsf{JSh}}(P_0, P_2, g^2)$ and $\Pi_{\mathsf{JSh}}(P_0, P_3, g)$ to generate $[\![g^1]\!]$, $[\![g^2]\!]$ and $[\![g]\!]$, respectively.
4. Parties robustly reconstruct $g - g^1 - g^2$ and check equality to 0. If it is 0, then parties continue with rest of the computation. Else, P_0 is identified to be corrupt and $\mathsf{TTP} = P_1$.

Figure 2: Verification P_0's communication in the multiplication protocol of Tetrad-R$^{\mathsf{I}}$

Tetrad-R$^{\mathsf{II}}$. This variant (Fig. 12) avoids computation over the extended ring at the cost of communicating 1 extra ring element in the preprocessing, compared to Tetrad-R$^{\mathsf{I}}$. Note that the communication cost of this protocol is similar to that of the one in SWIFT [33]. We were unable to extend the latter's efficiently to support multi-input multiplication. Hence, we design Tetrad-R$^{\mathsf{II}}$ that has the same communication complexity as SWIFT but also supports multi-input multiplication, as well as truncation without any overhead. In order to get rid of Π_{VrfyP0}, the communication of w from P_0 to P_3 is split into 2 parts. (P_0, P_1) and (P_0, P_2) compute w in parts, and send them to P_3 using jsnd. This modification allows P_3 to compute $\mathsf{y}_1 + \mathsf{s}_1$ and $\mathsf{y}_2 + \mathsf{s}_2$ separately in the online phase. In addition, to enable P_2 to obtain y_1, P_1, P_3 can jsnd $\mathsf{y}_1 + \mathsf{s}_1$ to P_2. P_1 obtains $\mathsf{y}_2 + \mathsf{s}_2$ similarly.

3.3 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. 13.

4 Mixed Protocol Framework

Preliminary details about the garbling scheme are described in §D.1, and elaborate details are given in §D.

Garbled world. 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.

Instantiating the garbled world using existing 4PC GC-based protocols [13, 29] turn out to be overkill, as they are standalone protocols. For instance, [29] provides robust protocols by communicating 12 GCs while [13] requires generating and exchanging commitments on the inputs to ensure input consistency. On the other hand, the inputs to our protocol are consistent (due to $[\![\cdot]\!]$-sharing), and we do not need an explicit reconstruction, making it lighter overall.

Towards this, we propose 2 GC protocols – one requiring communication of 2 GC evaluations and 1 online round, and the other one requiring 1 GC and 2 rounds. Moreover, these protocols leverage the benefit of amortization which comes from using jsnd. 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. The 1 GC variant comprises of a single execution with Φ_1 acting as garblers and P_1 as the evaluator.

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 [31, 32], half gates [27, 58], and fixed AES-key [7] are deployed in our protocol.

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.

Input Phase. Given that the function input x is already available as $[\![\mathsf{x}]\!]^{\mathbf{B}}$, the boolean values $\mathsf{m_x}, \alpha_{\mathsf{x}}, \lambda_{\mathsf{x}}^3$, where $\alpha_{\mathsf{x}} = \lambda_{\mathsf{x}}^1 \oplus \lambda_{\mathsf{x}}^2$ and $\mathsf{x} = \mathsf{m_x} \oplus \alpha_{\mathsf{x}} \oplus \lambda_{\mathsf{x}}^3$, 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 $(\mathsf{m_x}, \alpha_{\mathsf{x}}, \lambda_{\mathsf{x}}^3)$ 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, $[\![\mathsf{x}]\!]^{\mathbf{C}} = ([\![\mathsf{m_x}]\!]^{\mathbf{G}}, [\![\alpha_{\mathsf{x}}]\!]^{\mathbf{G}}, [\![\lambda_{\mathsf{x}}^3]\!]^{\mathbf{G}})$ 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.

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 $[\![\mathsf{v}]\!]_i^{\mathbf{G}} = (\mathsf{K}_{\mathsf{v}}^{0,1}, \mathsf{K}_{\mathsf{v}}^{0,2})$, P_1 holds $[\![\mathsf{v}]\!]_1^{\mathbf{G}} = (\mathsf{K}_{\mathsf{v}}^{\mathsf{v},1}, \mathsf{K}_{\mathsf{v}}^{0,2})$ and P_2 holds $[\![\mathsf{v}]\!]_2^{\mathbf{G}} = (\mathsf{K}_{\mathsf{v}}^{0,1}, \mathsf{K}_{\mathsf{v}}^{\mathsf{v},2})$. Here, $\mathsf{K}_{\mathsf{v}}^{\mathsf{v},j} = \mathsf{K}_{\mathsf{v}}^{0,j} \oplus \mathsf{v}\Delta^j$ 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 $(\mathsf{m_x}, \alpha_{\mathsf{x}}, \lambda_{\mathsf{x}}^3)$, which are as defined above, is $[\![\cdot]\!]^{\mathbf{G}}$-shared. We write $[\![\mathsf{x}]\!]^{\mathbf{C}} = ([\![\mathsf{m_x}]\!]^{\mathbf{G}}, [\![\alpha_{\mathsf{x}}]\!]^{\mathbf{G}}, [\![\lambda_{\mathsf{x}}^3]\!]^{\mathbf{G}})$.

Generation of $[\![\mathsf{v}]\!]^{\mathbf{G}}$ and $[\![\mathsf{x}]\!]^{\mathbf{C}}$ Protocol $\Pi_{\mathsf{Sh}}^{\mathbf{G}}(\mathcal{P}, \mathsf{v})$ (Fig. 19) 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 $\{\mathsf{K}_{\mathsf{v}}^{\mathsf{b},1}\}_{\mathsf{b} \in \{0,1\}}$ which denotes the key for value b on wire v, following the free-XOR technique [31, 32]. P_k, P_l jsnd $\mathsf{K}_{\mathsf{v}}^{\mathsf{v},1}$ 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 $\{\mathsf{K}_{\mathsf{v}}^{\mathsf{b},2}\}_{\mathsf{b} \in \{0,1\}}$ while the evaluator P_2 holds $\mathsf{K}_{\mathsf{v}}^{\mathsf{v},2}$. Following this, the shares $[\![\mathsf{v}]\!]_s^{\mathbf{G}}$ held by $P_s \in \mathcal{P}$ are defined as $[\![\mathsf{v}]\!]_0^{\mathbf{G}} = [\![\mathsf{v}]\!]_3^{\mathbf{G}} = (\mathsf{K}_{\mathsf{v}}^{0,1}, \mathsf{K}_{\mathsf{v}}^{0,2})$, $[\![\mathsf{v}]\!]_1^{\mathbf{G}} = (\mathsf{K}_{\mathsf{v}}^{\mathsf{v},1}, \mathsf{K}_{\mathsf{v}}^{0,2})$, $[\![\mathsf{v}]\!]_2^{\mathbf{G}} = (\mathsf{K}_{\mathsf{v}}^{0,1}, \mathsf{K}_{\mathsf{v}}^{\mathsf{v},2})$.

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

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.

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 $\Pi_{\mathsf{Sh}}^{\mathbf{G}}$. 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.

Other Conversions.

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. 22. Boolean addition, when instantiated using the adder of ABY2.0 [45], requires $\log_4(\ell)$ rounds.

Boolean to Arithmetic. To convert a boolean sharing of v into an arithmetic sharing, we use techniques from [15, 33]. For a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, note that

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

where $\lambda_\mathsf{v}{}_i^{\mathsf{R}}, \mathsf{m_v}{}_i^{\mathsf{R}}$ 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. §5). 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. 23.

5 Building Blocks

We provide the details of the primitives needed for the applications in this section. Elaborate details appear in §C.

Dot Product (Scalar Product). Given $[\![\vec{\mathbf{a}}]\!],[\![\vec{\mathbf{b}}]\!]$ with $|\vec{\mathbf{a}}| = |\vec{\mathbf{b}}| = \mathsf{d}$, protocol Π_{dotp} (Fig. 3) computes $[\![\mathsf{z}]\!]$ such that $\mathsf{z} = (\vec{\mathbf{a}} \odot \vec{\mathbf{b}})^{\mathsf{t}}$ if truncation is enabled, else $\mathsf{z} = \vec{\mathbf{a}} \odot \vec{\mathbf{b}}$. Following [15, 33], 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 protocols for robust setting follows similarly from Tetrad-R$^{\mathsf{I}}$ and Tetrad-R$^{\mathsf{II}}$.

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

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

Preprocessing:

1. Parties locally compute the following:

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

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

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

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 $\mathsf{u}^1+\mathsf{u}^2 = \gamma^3_{\vec{\mathbf{a}}\vec{\mathbf{b}}} + \mathsf{r}$ for a random $\mathsf{r} \in_R \mathbb{Z}_{2^\ell}$.
3. P_0,P_3 compute $\mathsf{r} = \mathsf{u}^1+\mathsf{u}^2-\gamma^3_{\vec{\mathbf{a}}\vec{\mathbf{b}}}$ 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 $\mathsf{w} = \gamma^1_{\vec{\mathbf{a}}\vec{\mathbf{b}}} + \gamma^2_{\vec{\mathbf{a}}\vec{\mathbf{b}}} + \mathsf{s}$ 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. Parties locally compute the following:

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

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

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

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 3: Dot Product with / without Truncation.

Matrix multiplication is an extension of the dot product protocol. We abuse notation and follow the $[\![\cdot]\!]$-sharing semantics (ref. §2) 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.

Multi-input Multiplication. Inspired from ABY2.0 [45], 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 [45], 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 §3.2), 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. 1), 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. 14.

Analogously, $\Pi^{\mathsf{R}}_{\mathsf{Mult}}$ can be extended to support 3-input multiplication while costing 12 elements communication in preprocessing. 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 $\gamma_\mathsf{ab}, \gamma_\mathsf{ac}, \gamma_\mathsf{ad}, \gamma_\mathsf{bc}, \gamma_\mathsf{bd}, \gamma_\mathsf{cd}, \gamma_\mathsf{abc}, \gamma_\mathsf{acd}, \gamma_\mathsf{bcd}, \gamma_\mathsf{abcd}$ needs to be generated in the preprocessing. The protocol is very similar to the 3-input case, and the details are deferred to §C.

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 $\mathsf{v} = \mathsf{a} - \mathsf{b}$ [33, 41, 46].

To compute the msb, we use two variants - i) the communication optimized parallel prefix adder (PPA) circuit from ABY3 [41] ($2(\ell-1)$ AND gates, $\log \ell$ depth), and ii) the round optimized bit extraction circuit from ABY2 [45]. 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 $\mathsf{v} = (\mathsf{m_v} - \lambda^3_\mathsf{v}) + (-\lambda^1_\mathsf{v} - \lambda^2_\mathsf{v})$ as per the sharing semantics (cf. Table 3). P_0, P_3 execute $\Pi^{\mathbf{B}}_{\mathsf{JSh}}$ on $(-\lambda^1_\mathsf{v} - \lambda^2_\mathsf{v})$ during the preprocessing, while P_0, P_3 execute $\Pi^{\mathbf{B}}_{\mathsf{JSh}}$ on $(\mathsf{m_v} - \lambda^3_\mathsf{v})$ during the online phase to generate the respective boolean sharing.

Bit to Arithmetic. Protocol $\Pi_{\mathsf{bit2A}}([\![\mathsf{b}]\!]^{\mathbf{B}})$ (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 $\mathsf{b}^\mathsf{R} = (\mathsf{b}^\mathsf{R}_1 - \mathsf{b}^\mathsf{R}_2)^2$.

Let $\mathsf{b}_1 = \mathsf{m_b} \oplus \lambda^3_\mathsf{v}$ and $\mathsf{b}_2 = \lambda^1_\mathsf{v} \oplus \lambda^2_\mathsf{v}$. To compute $[\![\mathsf{b}]\!]$, a pair of parties can generate the arithmetic sharing corresponding to b^R_1 and b^R_2 by executing Π_{JSh}. $[\![\mathsf{b}]\!]$ can be computed by invoking Π_{Mult} once with inputs $\mathsf{x} = \mathsf{y} = \mathsf{b}^\mathsf{R}_1 - \mathsf{b}^\mathsf{R}_2$.

Using the techniques from [15, 33], we obtain a communication-optimized variant by trading off computation in the preprocessing. For this, note that

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

Let $\mathsf{v} = \mathsf{m}^\mathsf{R}_\mathsf{b}$ 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 the 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 $[\![\mathsf{b}]\!]$.

Piecewise Polynomials. Piece-wise polynomial functions are constructed as a series of constant polynomials $f_1, \ldots, f_m$ with public coefficients and $c_1 < \ldots < c_m$ such that,

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

For computing f, we first compute a set of bits $\mathsf{b}_1, \ldots, \mathsf{b}_m$ such that $\mathsf{b}_i = 1$ if $y \ge c_i$ and 0 otherwise. f can be computed as, $f(y) = \sum_{i=1}^m \mathsf{b}_i \cdot (f_i - f_{i-1})$, where $f_0 = 0$ and $f_m = 1$. 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. The bit injection protocol of [33] allows computing $[\![\mathsf{b}\cdot\mathsf{v}]\!]$ given $[\![\mathsf{b}]\!]^{\mathbf{B}}$ and $[\![\mathsf{v}]\!]$. $f(y)$ can be viewed as a sum of m bit injections, which results in the online communication being independent of m.

For ease of presentation, let $\mathsf{z} = \sum_{i=1}^m \mathsf{b}^\mathsf{R}_i \cdot \mathsf{v_i}$, where $\mathsf{v_i} \in \mathbb{Z}_{2^\ell}$, $\mathsf{b}_i \in \mathbb{Z}_2$ and $\mathsf{b}^\mathsf{R} \in \mathbb{Z}_{2^\ell}$ denotes the value b in $\mathbb{Z}_{2^\ell}$. Given $[\![\mathsf{b}_i]\!]^{\mathbf{B}}$ and $\mathsf{v_i}$ for $i \in \{1, 2, \ldots, m\}$, ($\Pi_{\mathsf{piecewise}}$, Fig. 17) generates $[\![\mathsf{z}]\!]$. Consider one term, $\mathsf{b}^\mathsf{R}\mathsf{v}$ in the expression for z. This can be written as follows.

$$\mathsf{b}^\mathsf{R}\mathsf{v} = (\mathsf{m_b} \oplus \lambda_\mathsf{b})^\mathsf{R}(\mathsf{m_v} - \lambda_\mathsf{v}) = (\mathsf{m}^\mathsf{R}_\mathsf{b} + \lambda^\mathsf{R}_\mathsf{b} - 2\mathsf{m}^\mathsf{R}_\mathsf{b}\lambda^\mathsf{R}_\mathsf{b})(\mathsf{m_v} - \lambda_\mathsf{v})$$

Thus, $\mathsf{z} = \sum_{i=1}^m \mathsf{b}^\mathsf{R}_i \cdot \mathsf{v_i}$ can be written as

$$\mathsf{z} = \sum_{i=1}^m \mathsf{m}^\mathsf{R}_{\mathsf{b}_i}\mathsf{m}_{\mathsf{v_i}} - \mathsf{m}^\mathsf{R}_{\mathsf{b}_i}\lambda_{\mathsf{v_i}} + (2\mathsf{m}^\mathsf{R}_{\mathsf{b}_i} - 1)(\lambda^\mathsf{R}_{\mathsf{b}_i}\lambda_{\mathsf{v_i}} - \mathsf{m}_{\mathsf{v_i}}\lambda^\mathsf{R}_{\mathsf{b}_i})$$

To compute $[\![z]\!]$, we let P_0 generate $\langle\cdot\rangle$-sharing of $\lambda^{R}_{b_i}\lambda_{v_i}$ and $\lambda^{R}_{b_i}$ for $i \in [m]$, where the correctness of the sharing is verified, similar to Π_{bit2A}. Note that the correctness for all $i \in [m]$ can be clubbed in a single check. Then, 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 z, generate the corresponding $[\![\cdot]\!]$-sharing using Π_{JSh}, and locally add these shares to obtain the $[\![\cdot]\!]$-sharing of z.

Non-linear activation functions, such as Rectified Linear Unit and Sigmoid, can be viewed as instantiations of piecewise polynomial functions as shown in ABY3 [41].

Oblivious Selection: Given $[\![\cdot]\!]$-shares of $x_0, x_1 \in \mathbb{Z}_{2^\ell}$ and $[\![b]\!]^{\mathbf{B}}$ where $b \in \{0,1\}$, oblivious selection (Π_{obv}) enables parties to generate re-randomized $[\![\cdot]\!]$-shares of $z = x_b$. Note that $z = b(x_1 - x_0) + x_0$ and can be computed using the piecewise polynomial protocol.

ArgMin/ ArgMax. Protocol Π_{argmin} (Fig. 18) allows parties to compute the index of the smallest element in a vector $\vec{\mathbf{x}} = (x_1, \ldots, x_m)$ of m elements, where $\vec{\mathbf{x}}$ is $[\![\cdot]\!]$-shared, i.e. each element $x_i \in \mathbb{Z}_{2^\ell}$ of $\vec{\mathbf{x}}$ is $[\![\cdot]\!]$-shared. The protocol outputs a $[\![\cdot]\!]^{\mathbf{B}}$-shared bit vector $\vec{\mathbf{b}}$ of size m which has a 1 at the index associated with the minimum value in $\vec{\mathbf{x}}$, and 0 elsewhere. We follow the standard tree-based approach [18] to recursively find the minimum value in $\vec{\mathbf{x}}$ while also updating $\vec{\mathbf{b}}$ to reflect the index of this smallest element. Each bit of $\vec{\mathbf{b}}$ is initialized to 1. The elements of $\vec{\mathbf{x}}$ are grouped into pairs and securely compared to find their pairwise minimum. Using this information, $\vec{\mathbf{b}}$ is updated such that b_j's are reset to 0 for x_j's $\in \vec{\mathbf{x}}$ which do not form the minimum in their respective pair; the other bits in $\vec{\mathbf{b}}$ still equal 1. The protocol recurses on the remaining elements $x_j \in \vec{\mathbf{x}}$, which were the pairwise minimums. Eventually, only one $b_j \in \vec{\mathbf{b}}$ equals 1, indicating that x_j is the minimum, with index j. Computing Π_{argmax} can be done similarly.

6 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 [36] and CIFAR-10 [34] dataset. Benchmarks of the protocols are against the state-of-the-art 4PC of Trident [15] and SWIFT [33] 4PC (supports only inference).

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 pairwise authenticated bidirectional synchronous channels (eg. instantiated via TLS over TCP/IP). We use a bandwidth of 40 MBps 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 [16] 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 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.

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 4: Benchmarking parameters

Benchmarking Parameters. We evaluate the protocols across a variety of parameters as given in Table 4. In addition to parameters such as runtime, communication, and *online throughput* (TP) [5,6,15,23,33,41,41], we report the cumulative runtime (sum of the up-time of all the hired servers). The reason behind doing so is that 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) [40] 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 parallel execution threads, each with roles of the servers changed. Load balancing has been performed in all the protocols benchmarked.

[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.

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 network. We refer readers to [43, 56] 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 [41, 46].
- *NN-2:* Convolutional neural network comprising of 2 hidden layers, with 100 and 10 nodes [15, 41, 49].
- *NN-3:* LeNet [35], comprises of 2 convolutional and fully connected layers, followed by maxpool for convolutional layers. This has approximately 431K parameters.
- *NN-4:* VGG16 [53] has 16 layers in total and contains fully-connected, convolutional, ReLU activation and maxpool layers. This has ≈ 138 million parameters.

Datasets. We use the following datasets:

- MNIST [36] 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 [34] 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 6000 images in each class. We evaluate NN-2, NN-4 on this dataset.

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. Usage of multi-input multiplication gates and the 2 GC variant of the garbled make 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).

$\mathsf{Tetrad}_\mathsf{T}$ and $\mathsf{Tetrad\text{-}R}_\mathsf{T}$ make use of multi-input multiplication gates and the 2 GC variant of the garbled world and are the fastest variants of the framework. On the other hand, $\mathsf{Tetrad}_\mathsf{C}$ and $\mathsf{Tetrad\text{-}R}_\mathsf{C}$ are variants with a minimal monetary cost. For robustness, we report only the numbers for $\mathsf{Tetrad\text{-}R}^{\mathsf{II}}$ and not $\mathsf{Tetrad\text{-}R}^{\mathsf{I}}$. This is because the overhead of $\mathsf{Tetrad\text{-}R}^{\mathsf{I}}$ over its fair counterpart Tetrad is very minimal for deep networks, like those considered in this work.

6.1 ML Training

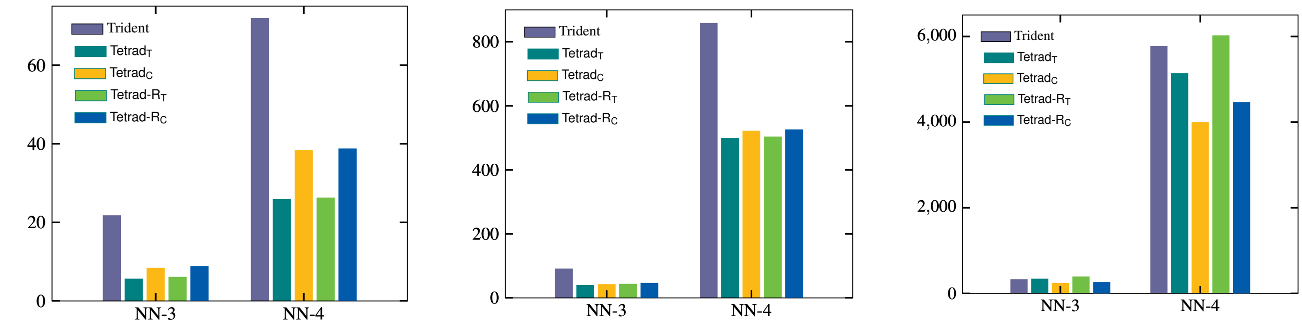
For training we consider NN-1, NN-2, NN-3 and NN-4 networks. We report values corresponding to one iteration, that comprises of a forward propagation followed by a backward propagation. More details are provided in §F.

Algo	Parameter	Trident	$\mathsf{Tetrad}_\mathsf{T}$	$\mathsf{Tetrad}_\mathsf{C}$	$\mathsf{Tetrad\text{-}R}_\mathsf{T}$	$\mathsf{Tetrad\text{-}R}_\mathsf{C}$
NN-1	PT_on	8.06	1.93	2.55	2.37	2.99
	PT_tot	10.76	5.05	5.27	5.84	6.26
	CT_tot	27.90	12.69	11.22	16.46	14.99
	$\mathsf{Comm}_\mathsf{tot}$	0.16	0.30	0.16	0.31	0.16
	Cost	49.33	58.51	34.29	62.27	37.77
	TP	1904.79	3792.64	3725.49	3792.63	3725.49
NN-2	PT_on	8.13	2.05	2.67	2.48	3.11
	PT_tot	11.47	5.79	6.14	6.58	7.13
	CT_tot	30.88	14.82	13.40	18.63	17.18
	$\mathsf{Comm}_\mathsf{tot}$	0.28	0.39	0.24	0.42	0.26
	Cost	70.00	75.67	49.16	81.93	54.31
	TP	428.16	652.75	644.69	652.75	644.69
NN-3	PT_on	21.79	5.67	8.40	6.11	8.84
	PT_tot	30.66	15.14	17.87	16.13	18.86
	CT_tot	91.68	40.01	42.76	43.78	46.53
	$\mathsf{Comm}_\mathsf{tot}$	1.59	1.94	1.28	2.25	1.40
	Cost	331.01	343.73	240.41	395.95	262.70
	TP	53.62	55.71	54.13	55.71	54.13
NN-4	PT_on	72.01	25.90	38.35	26.30	38.79
	PT_tot	283.89	182.13	194.58	183.08	195.57
	CT_tot	859.09	500.13	522.32	503.90	526.09
	$\mathsf{Comm}_\mathsf{tot}$	31.59	29.52	22.24	35.01	25.16
	Cost	5779.27	5146.10	3999.30	6025.79	4468.37
	TP	2.55	2.61	2.56	2.61	2.56

Table 5: 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.

Starting with the time-optimized variants ($\mathsf{Tetrad}_\mathsf{T}$, $\mathsf{Tetrad\text{-}R}_\mathsf{T}$) are $3-4\times$ 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 ReLU activation function using multi-input AND gates (cf. §5). Looking at the total communication ($\mathsf{Comm}_\mathsf{tot}$) in Table 5, we observe that the gap in $\mathsf{Comm}_\mathsf{tot}$ between $\mathsf{Tetrad}_\mathsf{T}$, $\mathsf{Tetrad\text{-}R}_\mathsf{T}$ 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 $\mathsf{Tetrad}_\mathsf{T}$ over Trident. Another reason is the there are two active parties (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.

The cost-optimized variants ($\mathsf{Tetrad}_\mathsf{C}$, $\mathsf{Tetrad\text{-}R}_\mathsf{C}$) on the other hand, are $1.5\times$ slower in the online phase compared to $\mathsf{Tetrad}_\mathsf{T}, \mathsf{Tetrad\text{-}R}_\mathsf{T}$. However, they are still faster than Trident



(a) Online Execution Time ($\mathsf{PT}_{\mathsf{on}}$) (b) Cumulative Time ($\mathsf{CT}_{\mathsf{tot}}$) (c) Monetary Cost ($\mathsf{Cost}$)

Figure 4: Training of NN-3 and NN-4: in terms of $\mathsf{PT}_{\mathsf{on}}$, $\mathsf{CT}_{\mathsf{tot}}$, and Cost (cf. Table 4)

owing to the rtt setup, as discussed above. When it comes to monetary cost, these variants are up to $20-40\%$ cheaper than their time-optimized counterparts and cheaper by around 30% over Trident.

These trends can be better captured with a pictorial representation as given in Figure 4 and Figure 24 (cf. §F).

6.2 ML Inference

We benchmark the inference phase of SVM and the aforementioned NNs. Training phase of SVM requires additional tools and primitives, and is out of scope of this work.

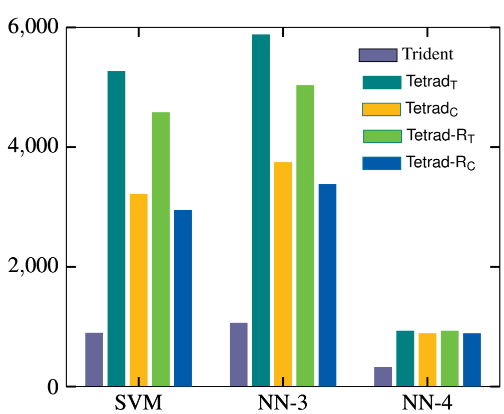


Figure 5: Inference of SVM, NN-3 and NN-4: in terms of TP

Similar to training, the time-optimized variants for inference are faster when it comes to $\mathsf{PT}_{\mathsf{on}}$, 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 5. 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 6 which shows that $\mathsf{Tetrad}_\mathsf{T}, \mathsf{Tetrad\text{-}R}_\mathsf{T}$ save on monetary cost up to a factor of 6 over Trident.

Algo	Parameter	Trident	Tetrad$_T$	Tetrad$_C$	Tetrad-R$_T$	Tetrad-R$_C$
SVM	$\mathsf{PT}_{\mathsf{on}}$	17.09	2.91	4.77	3.35	5.21
	$\mathsf{PT}_{\mathsf{tot}}$	17.37	3.19	5.05	4.18	6.04
	$\mathsf{CT}_{\mathsf{tot}}$	47.02	6.99	10.70	10.76	14.47
	$\mathsf{Comm}_{\mathsf{tot}}$	1.36	2.34	1.25	2.84	1.36
	Cost	39.92	6.26	9.23	9.53	12.43
	TP	898.80	5271.74	3221.29	4581.56	2949.76
NN-1	$\mathsf{PT}_{\mathsf{on}}$	5.87	1.31	1.87	1.75	2.31
	$\mathsf{PT}_{\mathsf{tot}}$	6.15	1.58	2.14	2.57	3.13
	$\mathsf{CT}_{\mathsf{tot}}$	16.75	3.76	4.88	7.54	8.65
	$\mathsf{Comm}_{\mathsf{tot}}$	0.06	0.09	0.05	0.11	0.06
	Cost	14.15	3.19	4.13	6.38	7.32
	TP	2615.35	11734.60	8226.93	8787.84	6661.00
NN-2	$\mathsf{PT}_{\mathsf{on}}$	5.87	1.31	1.87	1.75	2.31
	$\mathsf{PT}_{\mathsf{tot}}$	6.15	1.58	2.14	2.57	3.13
	$\mathsf{CT}_{\mathsf{tot}}$	16.75	3.77	4.88	7.54	8.66
	$\mathsf{Comm}_{\mathsf{tot}}$	0.26	0.37	0.22	0.45	0.24 (+0.01)
	Cost	14.19	3.24	4.16	6.44	7.35
	TP	2615.35	11734.60	8226.93	8787.84	6661.00
NN-3	$\mathsf{PT}_{\mathsf{on}}$	14.42	2.61	4.10	3.05	4.54
	$\mathsf{PT}_{\mathsf{tot}}$	14.71	2.91	4.39	3.89	5.38 (+.01)
	$\mathsf{CT}_{\mathsf{tot}}$	39.92	6.43	9.40	10.20	13.18
	$\mathsf{Comm}_{\mathsf{tot}}$	5.62	8.42	4.76	10.24	5.27 (+0.12)
	Cost	34.59	6.74	8.68	10.21	11.95 (+0.02)
	TP	1065.35	5882.44	3746.89	5035.93	3384.51
NN-4	$\mathsf{PT}_{\mathsf{on}}$	47.05	7.85	12.69	8.29	13.13
	$\mathsf{PT}_{\mathsf{tot}}$	47.61	8.44	13.28	9.42	14.27 (+0.06)
	$\mathsf{CT}_{\mathsf{tot}}$	129.41	17.77	27.46	21.55	31.23 (+0.12)
	$\mathsf{Comm}_{\mathsf{tot}}$	85.69	124.09	71.27	150.92	79.15 (+2.18)
	Cost	122.66	34.40	34.32	41.77	38.74 (+0.44)
	TP	326.46	934.34	891.19	934.34	891.19

Table 6: 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. Values for $\mathsf{Tetrad\text{-}R}_\mathsf{C}$ and SWIFT are similar and the overhead, if any, is indicated along with the values.

Note that the cost-optimized variants underperform in terms of monetary cost compared to $\mathsf{Tetrad}_\mathsf{T}, \mathsf{Tetrad\text{-}R}_\mathsf{T}$. This is because, as mentioned earlier, run time plays a bigger role in monetary cost than communication. Hence for inference, the time-optimized variants become the optimal choice.

References

[1] 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 D. Hofheinz and A. Rosen, editors, *TCC 2019, Part I*, volume 11891 of *LNCS*, pages 471–501. Springer, Heidelberg, Dec. 2019.

[2] 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.

[3] B. Alon, E. Omri, and A. Paskin-Cherniavsky. MPC with friends and foes. In D. Micciancio and T. Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 677–706. Springer, Heidelberg, Aug. 2020.

[4] 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*, abs/2012.05064, 2020.

[5] 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*, pages 843–862. IEEE Computer Society Press, May 2017.

[6] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016*, pages 805–817. ACM Press, Oct. 2016.

[7] 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*, pages 478–492. IEEE Computer Society Press, May 2013.

[8] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM CCS 2012*, pages 784–796. ACM Press, Oct. 2012.

[9] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In A. Boldyreva and D. Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 67–97. Springer, Heidelberg, Aug. 2019.

[10] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In L. Cavallaro, J. Kinder, X. Wang, and J. Katz, editors, *ACM CCS 2019*, pages 869–886. ACM Press, Nov. 2019.

[11] M. Byali, H. Chaudhari, A. Patra, and A. Suresh. FLASH: Fast and robust framework for privacy-preserving machine learning. *PoPETs*, 2020(2):459–480, Apr. 2020.

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

[13] M. Byali, A. Joseph, A. Patra, and D. Ravi. Fast secure computation for small population over the internet. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 677–694. ACM Press, Oct. 2018.

[14] 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.

[15] H. Chaudhari, R. Rachuri, and A. Suresh. Trident: Efficient 4PC framework for privacy preserving machine learning. In *NDSS 2020*. The Internet Society, Feb. 2020.

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

[17] A. Dalskov, D. Escudero, and M. Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. Cryptology ePrint Archive, Report 2020/1330, 2020. https://eprint.iacr.org/2020/1330.

[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*, pages 1102–1120. IEEE Computer Society Press, May 2019.

[19] I. Damgård, C. Orlandi, and M. Simkin. Yet another compiler for active security or: Efficient MPC over arbitrary rings. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 799–829. Springer, Heidelberg, Aug. 2018.

[20] 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.

[21] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In D. Micciancio and T. Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 823–852. Springer, Heidelberg, Aug. 2020.

[22] M. Fredrikson, S. Jha, and T. Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In I. Ray, N. Li, and C. Kruegel, editors, *ACM CCS 2015*, pages 1322–1333. ACM Press, Oct. 2015.

[23] J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In J.-S. Coron and J. B. Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 225–255. Springer, Heidelberg, Apr. / May 2017.

[24] O. Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, UK, 2004.

[25] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In A. Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[26] S. D. Gordon, S. Ranellucci, and X. Wang. Secure computation with low communication from cross-checking. In T. Peyrin and S. Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 59–85. Springer, Heidelberg, Dec. 2018.

[27] S. Gueron, Y. Lindell, A. Nof, and B. Pinkas. Fast garbling of circuits under standard assumptions. *Journal of Cryptology*, 31(3):798–844, July 2018.

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

[29] Y. Ishai, R. Kumaresan, E. Kushilevitz, and A. Paskin-Cherniavsky. Secure computation with minimal interaction, revisited. In R. Gennaro and M. J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 359–378. Springer, Heidelberg, Aug. 2015.

[30] M. Keller, V. Pastro, and D. Rotaru. Overdrive: Making SPDZ great again. In J. B. Nielsen and V. Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 158–189. Springer, Heidelberg, Apr. / May 2018.

[31] V. Kolesnikov, P. Mohassel, and M. Rosulek. FleXOR: Flexible garbling for XOR gates that beats free-XOR. In J. A. Garay and R. Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 440–457. Springer, Heidelberg, Aug. 2014.

[32] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, Heidelberg, July 2008.

[33] 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.

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

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

[36] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010.

[37] 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.

[38] E. Makri, D. Rotaru, N. P. Smart, and F. Vercauteren. EPIC: Efficient private image classification (or: Learning from the masters). In M. Matsui, editor, *CT-RSA 2019*, volume 11405 of *LNCS*, pages 473–492. Springer, Heidelberg, Mar. 2019.

[39] S. Mazloom, P. H. Le, S. Ranellucci, and S. D. Gordon. Secure parallel computation on national scale volumes of data. In S. Capkun and F. Roesner, editors, *USENIX Security 2020*, pages 2487–2504. USENIX Association, Aug. 2020.

[40] P. Miao, S. Patel, M. Raykova, K. Seth, and M. Yung. Two-sided malicious security for private intersection-sum with cardinality. In D. Micciancio and T. Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 3–33. Springer, Heidelberg, Aug. 2020.

[41] P. Mohassel and P. Rindal. ABY^3: A mixed protocol framework for machine learning. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 35–52. ACM Press, Oct. 2018.

[42] P. Mohassel, M. Rosulek, and Y. Zhang. Fast and secure three-party computation: The garbled circuit approach. In I. Ray, N. Li, and C. Kruegel, editors, *ACM CCS 2015*, pages 591–602. ACM Press, Oct. 2015.

[43] P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*, pages 19–38. IEEE Computer Society Press, May 2017.

[44] S. Ohata and K. Nuida. Communication-efficient (client-aided) secure two-party protocols and its application. In J. Bonneau and N. Heninger, editors, *FC 2020*, volume 12059 of *LNCS*, pages 369–385. Springer, Heidelberg, Feb. 2020.

[45] 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.

[46] A. Patra and A. Suresh. BLAZE: Blazing fast privacy-preserving machine learning. In *NDSS 2020*. The Internet Society, Feb. 2020.

[47] B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai. SpOT-light: Lightweight private set intersection from sparse OT extension. In A. Boldyreva and D. Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 401–431. Springer, Heidelberg, Aug. 2019.

[48] P. Pullonen and S. Siim. Combining secret sharing and garbled circuits for efficient private IEEE 754 floating-point computations. In M. Brenner, N. Christin, B. Johnson, and K. Rohloff, editors, *FC 2015 Workshops*, volume 8976 of *LNCS*, pages 172–183. Springer, Heidelberg, Jan. 2015.

[49] 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 J. Kim, G.-J. Ahn, S. Kim, Y. Kim, J. López, and T. Kim, editors, *ASIACCS 18*, pages 707–721. ACM Press, Apr. 2018.

[50] P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In B. K. Roy and W. Meier, editors, *FSE 2004*, volume 3017 of *LNCS*, pages 371–388. Springer, Heidelberg, Feb. 2004.

[51] D. Rotaru and T. Wood. MArBled circuits: Mixing arithmetic and Boolean circuits with active security. In F. Hao, S. Ruj, and S. Sen Gupta, editors, *INDOCRYPT 2019*, volume 11898 of *LNCS*, pages 227–249. Springer, Heidelberg, Dec. 2019.

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

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

[54] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Stealing machine learning models via prediction APIs. In T. Holz and S. Savage, editors, *USENIX Security 2016*, pages 601–618. USENIX Association, Aug. 2016.

[55] S. Wagh, D. Gupta, and N. Chandran. SecureNN: 3-party secure computation for neural network training. *PoPETs*, 2019(3):26–49, July 2019.

[56] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *PoPETs*, 2021(1):188–208, Jan. 2021.

[57] A. C.-C. Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, Nov. 1982.

[58] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In E. Oswald and M. Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Heidelberg, Apr. 2015.

A Preliminaries

A.1 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 [41] 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 [6] and the latter on [5,23]. ASTRA [14] improved upon the 3PC of [5,6,23] by proposing faster protocols for the online phase with active security. As a result, secure inference of ASTRA is faster than ABY3. Building on [9], BLAZE [46] 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 [9]. SWIFT (3PC) improved upon BLAZE by using the distributed zero-knowledge protocol of [10], thereby achieving GOD. In an orthogonal line of work, FALCON [56] 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, [26] explores protocols for the security notions of abort. Inspired by the theoretical GOD construction in [26], FLASH proposed practical protocols with GOD for secure inference. Trident [15] improved protocols (in terms of communication) compared to [26] 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, [39] improved over [26] 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 [33] 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 [17] 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.3.

In the regime of constant-round protocols, [42] 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 [29] presents a robust 4-party computation protocol (4PC) with GOD in 2-rounds (which is optimal) at the expense of 12 garbled circuits. Further, [13] 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 [28]. TASTY was a passively secure compiler supporting generation of protocols based on homomorphic encryption and garbled circuits. This was followed by ABY [20], which proposed a mixed protocol framework, also with passive security, combining the arithmetic, boolean and garbled worlds. The recent work of ABY2 [45] improves upon the ABY framework, providing a faster online phase with applications to PPML. The work of [21, 51] proposed efficient mixed world conversions for the case of n parties with a dishonest majority. Both works have active security, with [51] supporting the inference of SVMs, and [21] supporting neural network inference.

In the honest majority setting, ABY3 [41] 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 [15] extended it to the 4-party setting.

A.2 Basic Primitives

Shared Key Setup. Let $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \rightarrow X$ 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 servers.

- 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 servers, 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 servers 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 server 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 6: Ideal functionality for shared-key setup

The key setup is modelled via a functionality $\mathcal{F}_{\text{SETUP}}$ (Fig. 6) 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.

Collision-Resistant Hash Function [50]. . 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 $m = \mathsf{poly}(\kappa)$.

A.3 Security Model

We prove security using the real-world/ ideal-word simulation paradigm [24, 37]. 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.

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 $x'_i = \texttt{abort}$.

Output: If there exists an $i \in \{0,1,2,3\}$ such that $x'_i = \texttt{abort}$, 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 7: Fair functionality for computing function f

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

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 8: GOD functionality for computing function f

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 GOD appears in Fig. 7 and Fig. 8, respectively.

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 third party TTP 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 [3]. 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. [17], where the protocol does not demand the inputs be sent to a TTP. 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.

A.4 Comparison with Fantastic Four [17]

We analyse the performance of Fantastic Four [17] where execution proceeds in segments (cf. §6.4, [17]). 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.

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

Work	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$^{\mathsf{I}}$(on-demand)	-	5ℓ	3
Tetrad-R$^{\mathsf{II}}$(on-demand)	-	6ℓ	3

Table 7: Comparison with Fantastic Four [17]

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 variants of our robust protocols (Tetrad-R$^{\mathsf{I}}$, Tetrad-R$^{\mathsf{II}}$). The results are summarised in Table 7. We remark that the values for their cases are obtained from Table 1 of their paper [17].

B 4PC Protocol

Joint-send for robust protocols.

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 servers. 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 servers set $\mathsf{TTP} = P_l$ if $\mathsf{b}' = 1$, terminate otherwise.

Figure 9: Joint-Send for robust protocols

Lemma B.1 (Communication). *Protocol Π_{jsnd} (Fig. 9) 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. □

Sharing Protocol.

Lemma B.2 (Communication). *Protocol Π_{Sh} (Fig. 10) 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.2). 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. □

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

Preprocessing: Parties sample the following:

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

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 accepts a default value for the case of robust security.

Figure 10: $[\![\cdot]\!]$-sharing of a value v by party P_i.

Reconstruction Protocol.

Lemma B.3 (Communication). *Protocol Π_{Rec} (Fig. 11) 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. Parties compute $\mathsf{v} = \mathsf{m_v} - \lambda_{\mathsf{v}}^1 - \lambda_{\mathsf{v}}^2 - \lambda_{\mathsf{v}}^3$.

Figure 11: Reconstruction of value v among parties in $\mathcal{P}$.

Multiplication Protocol.

Lemma B.4 (Communication). *Protocol Π_{Mult} (Fig. 1) (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 §3.1. 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 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. $\square$

Robust Multiplication Protocol in Tetrad-R$^{\text{II}}$. The formal protocol for the robust multiplication in Tetrad-R$^{\text{II}}$, $\Pi^{\text{R}}_{\text{Mult}}$, appears in Fig. 12. The primary difference from the fair counterpart is that the communication of w from P_0 to P_3 in the preprocessing is now split into two parts. $(P_0, P_1), (P_0, P_2)$ communicates $\mathsf{w}_1, \mathsf{w}_2$ respectively to P_3 via jsnd.

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

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

Preprocessing:

1. Parties 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. P_0, P_3 compute $\mathsf{r} = \mathsf{u}^1 + \mathsf{u}^2 - \gamma^3_{\mathsf{ab}}$ 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 $\mathsf{s}_1, \mathsf{s}_2 \in_R \mathbb{Z}_{2^\ell}$. P_0, P_j jsnd $\mathsf{w}_j = \gamma^j_{\mathsf{ab}} + \mathsf{s}_j$ to P_3 for $j \in \{1,2\}$.

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

1. Parties locally compute the following:

$$P_1, P_3 : \mathsf{y}_1 + \mathsf{s}_1 = -\lambda^1_{\mathsf{a}}\mathsf{m}_{\mathsf{b}} - \lambda^1_{\mathsf{b}}\mathsf{m}_{\mathsf{a}} + \mathsf{u}^1 + \mathsf{w}_1$$
$$P_2, P_3 : \mathsf{y}_2 + \mathsf{s}_2 = -\lambda^2_{\mathsf{a}}\mathsf{m}_{\mathsf{b}} - \lambda^2_{\mathsf{b}}\mathsf{m}_{\mathsf{a}} + \mathsf{u}^2 + \mathsf{w}_2$$
$$P_1, P_2 : \mathsf{y}_3 = -\lambda^3_{\mathsf{a}}\mathsf{m}_{\mathsf{b}} - \lambda^3_{\mathsf{b}}\mathsf{m}_{\mathsf{a}}$$

2. P_1, P_3 jsnd $\mathsf{y}_1 + \mathsf{s}_1$ to P_2, while P_1, P_3 jsnd $\mathsf{y}_2 + \mathsf{s}_2$ to P_1.
3. P_1, P_2 locally compute $\mathsf{z} + \mathsf{r} = (\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{y}_3) + \mathsf{m}_{\mathsf{a}}\mathsf{m}_{\mathsf{b}} - \mathsf{s}_1 - \mathsf{s}_2$.
4. If $\mathsf{isTr} = 1$, P_1, P_2 locally 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 $[\![\mathsf{p}]\!]$.
5. Parties locally compute $[\![\mathsf{o}]\!] = [\![\mathsf{p}]\!] - [\![\mathsf{q}]\!]$. Here $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and z otherwise.

Figure 12: Robust multiplication in Tetrad-R$^{\text{II}}$.

Lemma B.5 (Communication). *Protocol $\Pi^{\text{R}}_{\text{Mult}}$ (Fig. 12) (in Tetrad-R$^{\text{II}}$) requires 3ℓ bits of communication in the preprocessing phase, and 1 round and 3ℓ bits of communication in the online phase.*

Proof. During preprocessing, the 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 §3.1. In addition, P_0, P_j for $j \in \{1,2\}$ communicates w_j to P_3 via jsnd requiring additional 2ℓ bits. The online phase is similar to the fair multiplication protocol (Π_{Mult}) and the costs follow from Lemma B.4. $\square$

B.1 Function-independent preprocessing

We provide the fair multiplication, $\Pi^{\text{NoPre}}_{\text{Mult}}$, for *function-independent* preprocessing in Fig. 13. 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. 13) can be performed along with the communication among P_1, P_2 (Step 4 in Fig. 13) 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 $\Pi^{\text{NoPre}}_{\text{Mult}}(\mathsf{a}, \mathsf{b}, \mathsf{isTr})$

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

Online:

1. Parties 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 $\mathsf{y} = (\mathsf{z} + \mathsf{r}) - \mathsf{m}_{\mathsf{a}}\mathsf{m}_{\mathsf{b}}$. Parties locally compute the following:

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

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: $\mathsf{r} = \mathsf{u}^1 + \mathsf{u}^2 - \gamma^3_{\mathsf{ab}}$; $\mathsf{q} = \mathsf{r}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$, else $\mathsf{q} = \mathsf{r}$; Execute $\Pi_{\mathsf{JSh}}(P_0, P_3, \mathsf{q})$.
   (b) P_1, P_2: $\mathsf{z} + \mathsf{r} = (\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{y}_3) + \mathsf{m}_{\mathsf{a}}\mathsf{m}_{\mathsf{b}}$; $\mathsf{p} = (\mathsf{z} + \mathsf{r})^{\mathsf{t}}$ if $\mathsf{isTr} = 1$, else $\mathsf{p} = \mathsf{z} + \mathsf{r}$; Execute $\Pi_{\mathsf{JSh}}(P_1, P_2, \mathsf{p})$.
6. Parties locally compute $[\![\mathsf{o}]\!] = [\![\mathsf{p}]\!] - [\![\mathsf{q}]\!]$. Here $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and z otherwise.

Verification:

1. P_0, P_1, P_2 sample random $\mathsf{s} \in_R \mathbb{Z}_{2^\ell}$. P_0 sends $\mathsf{w} = \gamma^1_{\mathsf{ab}} + \gamma^2_{\mathsf{ab}} + \mathsf{s}$ to P_3.
2. P_3 computes $\mathsf{v} = -(\lambda^1_{\mathsf{a}} + \lambda^2_{\mathsf{a}})\mathsf{m}_{\mathsf{b}} - (\lambda^1_{\mathsf{b}} + \lambda^2_{\mathsf{b}})\mathsf{m}_{\mathsf{a}} + \mathsf{u}^1 + \mathsf{u}^2 + \mathsf{w}$ 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 13: Fair multiplication without preprocessing.

C Building Blocks

Dot Product (Scalar Product).

Lemma C.1 (Communication)**.** *Protocol Π_{dotp} (Fig. 3) (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. 1) and the costs follow from Lemma B.4. □

Lemma C.2 (Communication)**.** *Protocol Π_{dotp} (Fig. 3) (in Tetrad-R^{II}) requires 3ℓ 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. 1) and the costs follow from Lemma B.5. □

Multi-input Multiplication.

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}:
   – Parties invoke $\mathcal{F}_{\mathsf{zero}}$ to enable P_0,P_j obtain Z_j for $j\in\{1,2,3\}$ such that $Z_1+Z_2+Z_3=0$.
   $$P_0,P_1 \text{ jsnd } (\gamma_{\mathsf{ab}})^1 = \lambda_{\mathsf{a}}^1\lambda_{\mathsf{b}}^3+\lambda_{\mathsf{a}}^3\lambda_{\mathsf{b}}^1+\lambda_{\mathsf{a}}^3\lambda_{\mathsf{b}}^3+Z_1 \text{ to } P_2.$$
   $$P_0,P_2 \text{ jsnd } (\gamma_{\mathsf{ab}})^2 = \lambda_{\mathsf{a}}^2\lambda_{\mathsf{b}}^3+\lambda_{\mathsf{a}}^3\lambda_{\mathsf{b}}^2+\lambda_{\mathsf{a}}^2\lambda_{\mathsf{b}}^2+Z_2 \text{ to } P_3.$$
   $$P_0,P_3 \text{ jsnd } (\gamma_{\mathsf{ab}})^3 = \lambda_{\mathsf{a}}^1\lambda_{\mathsf{b}}^2+\lambda_{\mathsf{a}}^2\lambda_{\mathsf{b}}^1+\lambda_{\mathsf{a}}^1\lambda_{\mathsf{b}}^1+Z_3 \text{ to } P_1.$$
   – Set $\langle\gamma_{\mathsf{ab}}\rangle$ as $\gamma_{\mathsf{ab}}^1=(\!(\gamma_{\mathsf{ab}})\!)^3,\ \gamma_{\mathsf{ab}}^2=(\!(\gamma_{\mathsf{ab}})\!)^2,\ \gamma_{\mathsf{ab}}^3=(\!(\gamma_{\mathsf{ab}})\!)^1$.
2. Computation for γ_{ac}:
   – Parties locally compute the following:
   $$P_0,P_1 : \gamma_{\mathsf{ac}}^1 = \lambda_{\mathsf{a}}^1\lambda_{\mathsf{c}}^3+\lambda_{\mathsf{a}}^3\lambda_{\mathsf{c}}^1+\lambda_{\mathsf{a}}^3\lambda_{\mathsf{c}}^3$$
   $$P_0,P_2 : \gamma_{\mathsf{ac}}^2 = \lambda_{\mathsf{a}}^2\lambda_{\mathsf{c}}^3+\lambda_{\mathsf{a}}^3\lambda_{\mathsf{c}}^2+\lambda_{\mathsf{a}}^2\lambda_{\mathsf{c}}^2$$
   $$P_0,P_3 : \gamma_{\mathsf{ac}}^3 = \lambda_{\mathsf{a}}^1\lambda_{\mathsf{c}}^2+\lambda_{\mathsf{a}}^2\lambda_{\mathsf{c}}^1+\lambda_{\mathsf{a}}^1\lambda_{\mathsf{c}}^1$$
   – P_0,P_3 and P_1 sample random $\mathsf{u}_{\mathsf{ac}}^1\in_R\mathbb{Z}_{2^\ell}$. P_0,P_3 compute and jsnd $\mathsf{u}_{\mathsf{ac}}^2=\gamma_{\mathsf{ac}}^3-\mathsf{u}_{\mathsf{ac}}^1$ to P_2.
   – P_0,P_1,P_2 sample random $\mathsf{s}_{\mathsf{ac}}\in_R\mathbb{Z}_{2^\ell}$. P_0 sends $\mathsf{w}_{\mathsf{ac}}=\gamma_{\mathsf{ac}}^1+\gamma_{\mathsf{ac}}^2+\mathsf{s}_{\mathsf{ac}}$ to P_3.
3. Computation for γ_{bc}: Similar to Step 2 (for γ_{ac}). P_1,P_2 obtain $\mathsf{u}_{\mathsf{bc}}^1,\mathsf{u}_{\mathsf{bc}}^2$ respectively such that $\mathsf{u}_{\mathsf{bc}}^1+\mathsf{u}_{\mathsf{bc}}^2=\gamma_{\mathsf{bc}}^3$. P_3 obtains $\mathsf{w}_{\mathsf{bc}}=\gamma_{\mathsf{bc}}^1+\gamma_{\mathsf{bc}}^2+\mathsf{s}_{\mathsf{bc}}$.
4. Computation for γ_{abc}:
   – Using γ_{ab} (Step 1), λ_{c}, compute the following:
   $$P_0,P_1 : \gamma_{\mathsf{abc}}^1 = \gamma_{\mathsf{ab}}^1\lambda_{\mathsf{c}}^3+\gamma_{\mathsf{ab}}^3\lambda_{\mathsf{c}}^1+\gamma_{\mathsf{ab}}^3\lambda_{\mathsf{c}}^3$$
   $$P_0,P_2 : \gamma_{\mathsf{abc}}^2 = \gamma_{\mathsf{ab}}^2\lambda_{\mathsf{c}}^3+\gamma_{\mathsf{ab}}^3\lambda_{\mathsf{c}}^2+\gamma_{\mathsf{ab}}^2\lambda_{\mathsf{c}}^2$$
   $$P_0,P_3 : \gamma_{\mathsf{abc}}^3 = \gamma_{\mathsf{ab}}^1\lambda_{\mathsf{c}}^2+\gamma_{\mathsf{ab}}^2\lambda_{\mathsf{c}}^1+\gamma_{\mathsf{ab}}^1\lambda_{\mathsf{c}}^1$$
   – P_0,P_3 and P_j sample random $\mathsf{u}_{\mathsf{abc}}^j\in_R\mathbb{Z}_{2^\ell}$ for $j\in\{1,2\}$. Let $\mathsf{u}_{\mathsf{abc}}^1+\mathsf{u}_{\mathsf{abc}}^2=\gamma_{\mathsf{abc}}^3+\mathsf{r}$ for $\mathsf{r}\in_R\mathbb{Z}_{2^\ell}$.
   – P_0,P_1,P_2 sample random $\mathsf{s}\in_R\mathbb{Z}_{2^\ell}$. P_0 sends $\mathsf{w}_{\mathsf{abc}}=\gamma_{\mathsf{abc}}^1+\gamma_{\mathsf{abc}}^2+\mathsf{s}$ to P_3.
5. P_0,P_3 compute $\mathsf{r}=\mathsf{u}_{\mathsf{abc}}^1+\mathsf{u}_{\mathsf{abc}}^2-\gamma_{\mathsf{abc}}^3$ 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. Parties locally compute the following:
   $$P_1 : \mathsf{y}_1 = -\lambda_{\mathsf{a}}^1\mathsf{m}_{\mathsf{bc}}-\lambda_{\mathsf{b}}^1\mathsf{m}_{\mathsf{ac}}-\lambda_{\mathsf{c}}^1\mathsf{m}_{\mathsf{ab}}+\gamma_{\mathsf{ab}}^1\mathsf{m}_{\mathsf{c}} + (\gamma_{\mathsf{ac}}^1+\mathsf{u}_{\mathsf{ac}}^1)\mathsf{m}_{\mathsf{b}}+(\gamma_{\mathsf{bc}}^1+\mathsf{u}_{\mathsf{bc}}^1)\mathsf{m}_{\mathsf{a}}+(\gamma_{\mathsf{abc}}^1+\mathsf{u}_{\mathsf{abc}}^1)$$
   $$P_2 : \mathsf{y}_2 = -\lambda_{\mathsf{a}}^2\mathsf{m}_{\mathsf{bc}}-\lambda_{\mathsf{b}}^2\mathsf{m}_{\mathsf{ac}}-\lambda_{\mathsf{c}}^2\mathsf{m}_{\mathsf{ab}}+\gamma_{\mathsf{ab}}^2\mathsf{m}_{\mathsf{c}} + (\gamma_{\mathsf{ac}}^2+\mathsf{u}_{\mathsf{ac}}^2)\mathsf{m}_{\mathsf{b}}+(\gamma_{\mathsf{bc}}^2+\mathsf{u}_{\mathsf{bc}}^2)\mathsf{m}_{\mathsf{a}}+(\gamma_{\mathsf{abc}}^2+\mathsf{u}_{\mathsf{abc}}^2)$$
   $$P_1,P_2 : \mathsf{y}_3 = -\lambda_{\mathsf{a}}^3\mathsf{m}_{\mathsf{bc}}-\lambda_{\mathsf{b}}^3\mathsf{m}_{\mathsf{ac}}-\lambda_{\mathsf{c}}^3\mathsf{m}_{\mathsf{ab}}+\gamma_{\mathsf{ab}}^3\mathsf{m}_{\mathsf{c}}$$
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. Parties locally compute $[\![\mathsf{o}]\!]=[\![\mathsf{p}]\!]-[\![\mathsf{q}]\!]$. Here $\mathsf{o}=\mathsf{z}^t$ if $\mathsf{isTr}=1$ and z otherwise.
5. *Verification:*
   – Parties locally compute the following:
   $$P_3 : \mathsf{v} = -(\lambda_{\mathsf{a}}^1+\lambda_{\mathsf{a}}^2)\mathsf{m}_{\mathsf{bc}}-(\lambda_{\mathsf{b}}^1+\lambda_{\mathsf{b}}^2)\mathsf{m}_{\mathsf{ac}}-(\lambda_{\mathsf{c}}^1+\lambda_{\mathsf{c}}^2)\mathsf{m}_{\mathsf{ab}} + (\gamma_{\mathsf{ab}}^1+\gamma_{\mathsf{ab}}^2)\mathsf{m}_{\mathsf{c}}+(\mathsf{w}_{\mathsf{ac}}+\gamma_{\mathsf{ac}}^3)\mathsf{m}_{\mathsf{b}}+(\mathsf{w}_{\mathsf{bc}}+\gamma_{\mathsf{bc}}^3)\mathsf{m}_{\mathsf{a}} + (\mathsf{w}_{\mathsf{abc}}+\gamma_{\mathsf{abc}}^3+\mathsf{r})$$
   $$P_1,P_2 : \mathsf{v}' = \mathsf{y}_1+\mathsf{y}_2-\mathsf{s}_{\mathsf{ac}}\mathsf{m}_{\mathsf{b}}-\mathsf{s}_{\mathsf{bc}}\mathsf{m}_{\mathsf{a}}+\mathsf{s}$$
   – P_3 sends $\mathsf{H}(\mathsf{v})$ to P_1,P_2, who `abort` iff $\mathsf{H}(\mathsf{v})\neq\mathsf{H}(\mathsf{v}')$.

Figure 14: 3-input fair multiplication in Tetrad.

Lemma C.3 (Communication)**.** *Protocol Π_{Mult3} (Fig. 14) (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. 1). 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. □

Protocol $\Pi^{\mathsf{R}}_{\mathsf{Mult3}}(\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{isTr})$

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

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

   – Set $\langle\gamma_{\mathsf{ab}}\rangle$ as $\gamma^1_{\mathsf{ab}}=(\!(\gamma_{\mathsf{ab}})\!)^3,\ \gamma^2_{\mathsf{ab}}=(\!(\gamma_{\mathsf{ab}})\!)^2,\ \gamma^3_{\mathsf{ab}}=(\!(\gamma_{\mathsf{ab}})\!)^1$.
2. Computation for $\gamma_{\mathsf{ac}},\gamma_{\mathsf{bc}}$: Similar to Step 1 (for γ_{ab}).
3. 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 $\mathsf{s}_1,\mathsf{s}_2\in_R\mathbb{Z}_{2^\ell}$. P_0,P_j jsnd $\mathsf{w}^j=\gamma^j_{\mathsf{abc}}+\mathsf{s}_j$ to P_3 for $j\in\{1,2\}$.
4. P_0,P_3 compute $\mathsf{r}=\mathsf{u}^1_{\mathsf{abc}}+\mathsf{u}^2_{\mathsf{abc}}-\gamma^3_{\mathsf{abc}}$ and set $\mathsf{q}=\mathsf{r}^{\mathsf{t}}$ 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 $\mathsf{y}=(\mathsf{z}+\mathsf{r})-\mathsf{m}_{\mathsf{abc}}+\mathsf{s}_1+\mathsf{s}_2$.

1. Parties locally compute the following:

$$\begin{aligned}P_0,P_1:\mathsf{y}_1&=-\lambda^1_{\mathsf{a}}\mathsf{m}_{\mathsf{bc}}-\lambda^1_{\mathsf{b}}\mathsf{m}_{\mathsf{ac}}-\lambda^1_{\mathsf{c}}\mathsf{m}_{\mathsf{ab}}+\gamma^1_{\mathsf{ab}}\mathsf{m}_{\mathsf{c}}+\gamma^1_{\mathsf{ac}}\mathsf{m}_{\mathsf{b}}\\&\quad+\gamma^1_{\mathsf{bc}}\mathsf{m}_{\mathsf{a}}+\mathsf{u}^1_{\mathsf{abc}}+\mathsf{w}^1\\P_0,P_2:\mathsf{y}_2&=-\lambda^2_{\mathsf{a}}\mathsf{m}_{\mathsf{bc}}-\lambda^2_{\mathsf{b}}\mathsf{m}_{\mathsf{ac}}-\lambda^2_{\mathsf{c}}\mathsf{m}_{\mathsf{ab}}+\gamma^2_{\mathsf{ab}}\mathsf{m}_{\mathsf{c}}+\gamma^2_{\mathsf{ac}}\mathsf{m}_{\mathsf{b}}\\&\quad+\gamma^2_{\mathsf{bc}}\mathsf{m}_{\mathsf{a}}+\mathsf{u}^2_{\mathsf{abc}}+\mathsf{w}^2\\P_1,P_2:\mathsf{y}_3&=-\lambda^3_{\mathsf{a}}\mathsf{m}_{\mathsf{bc}}-\lambda^3_{\mathsf{b}}\mathsf{m}_{\mathsf{ac}}-\lambda^3_{\mathsf{c}}\mathsf{m}_{\mathsf{ab}}+\gamma^3_{\mathsf{ab}}\mathsf{m}_{\mathsf{c}}+\gamma^3_{\mathsf{ac}}\mathsf{m}_{\mathsf{b}}\\&\quad+\gamma^3_{\mathsf{bc}}\mathsf{m}_{\mathsf{a}}\end{aligned}$$

2. P_1,P_3 jsnd y_1 to P_2, while P_2,P_3 jsnd y_2 to P_1. P_1,P_2 locally compute $\mathsf{z}+\mathsf{r}=(\mathsf{y}_1+\mathsf{y}_2+\mathsf{y}_3)+\mathsf{m}_{\mathsf{abc}}-\mathsf{s}_1-\mathsf{s}_2$.
3. If $\mathsf{isTr}=1$, P_1,P_2 set $\mathsf{p}=(\mathsf{z}+\mathsf{r})^{\mathsf{t}}$, else $\mathsf{p}=\mathsf{z}+\mathsf{r}$. 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.

Figure 15: 3-input robust multiplication in Tetrad-R^{II}.

Lemma C.4 (Communication). *Protocol $\Pi^{\mathsf{R}}_{\mathsf{Mult3}}$ (Fig. 15) (in Tetrad) requires 12ℓ bits of communication in preprocessing, and 1 round and 3ℓ bits of communication in the online phase.*

Proof. In the preprocessing, computation of each of $\gamma_{\mathsf{ab}},\gamma_{\mathsf{ac}},\gamma_{\mathsf{bc}}$ involves three instances of jsnd. The computation of γ_{abc} is similar to the preprocessing of robust multiplication protocol (Fig. 12). The communication pattern of the online phase is similar to that of the robust multiplication protocol. The costs follow from Lemma B.5 and Lemma B.1. □

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

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

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.

Bit to Arithmetic. For verifying the $\langle\cdot\rangle$-sharing of u by P_0, we let P_3 obtain the bit $(\lambda_{\mathsf{b}}\oplus\mathsf{r}_b)$ as well as its arithmetic equivalent $(\lambda_{\mathsf{b}}\oplus\mathsf{r}_b)^{\mathsf{R}}$ 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 $(\lambda_{\mathsf{b}}\oplus\mathsf{r}_b)^{\mathsf{R}}$. 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.

Lemma C.5 (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. □

Protocol $\Pi_{\mathsf{bit2A}}(\llbracket \mathsf{b} \rrbracket^{\mathbf{B}})$

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

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 $\mathsf{r_b} \in \{0,1\}$ and $\mathsf{r} \in \mathbb{Z}_{2^\ell}$.
3. P_1, P_2 jsnd $\lambda_{\mathsf{b}}^3 \oplus \mathsf{r_b}$ to P_3. P_3 locally sets $\lambda_{\mathsf{b}} \oplus \mathsf{r_b} = (\lambda_{\mathsf{b}}^1 \oplus \lambda_{\mathsf{b}}^2) \oplus (\lambda_{\mathsf{b}}^3 \oplus \mathsf{r_b})$.
4. Parties compute: $P_1, P_0 : \mathsf{w}_1 = \mathsf{r}_{\mathsf{b}}^{\mathsf{R}} + (\mathsf{u}^1 + \mathsf{u}^3)(1 - 2\mathsf{r}_{\mathsf{b}}^{\mathsf{R}}) + \mathsf{r}$, $P_2, P_0 : \mathsf{w}_2 = (\mathsf{u}^2)(1 - 2\mathsf{r}_{\mathsf{b}}^{\mathsf{R}}) - \mathsf{r}$.
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 $\mathsf{TTP} = P_1$ (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 $\llbracket \cdot \rrbracket$-shares.
3. Compute $\llbracket \mathsf{y} \rrbracket = \llbracket \mathsf{y}_1 \rrbracket + \llbracket \mathsf{y}_2 \rrbracket + \llbracket \mathsf{y}_3 \rrbracket$.

Figure 16: Bit to Arithmetic conversion

Piecewise Polynomials. Without loss of generality, consider the case where $m = 1$. Similar to Π_{bit2A},

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

During the preprocessing, we let P_0 generate the $\langle \cdot \rangle$-shares of $(\lambda_{\mathsf{b}})^{\mathsf{R}}$ and $(\lambda_{\mathsf{b}})^{\mathsf{R}}\lambda_{\mathsf{v}}$. The correctness of the sharing is verified using techniques from Trident [15]. During the online phase, the communication corresponding to the m instances can be clubbed together resulting in a communication of just 3ℓ bits.

Lemma C.6 (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_{bit2A}. An exception is for the verification of $\langle \mu_i \rangle$ where its not needed to communicate a boolean bit to P_3 as for the case of $\langle \mathsf{u}_i \rangle$. The communication in the online phase is similar to that of the Π_{bit2A} protocol. The cost follows from Lemma C.5. □

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

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

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

1. Generation of $\langle \mathsf{u}_i \rangle, \langle \mu_i \rangle$: P_0, P_3, P_j for $j \in \{1,2\}$ sample u_i^j, μ_i^j. P_0 sends $\mathsf{u}_i^3 = \mathsf{u}_i - \mathsf{u}_i^1 - \mathsf{u}_i^2$ and $\mu_i^3 = \mu_i - \mu_i^1 - \mu_i^2$ to P_1, P_2.
2. Verifying correctness of $\langle \mathsf{u}_i \rangle$: Similar to the verification in the preprocessing of Π_{bit2A} (Fig. 16).
3. Verifying correctness of $\langle \mu \rangle_i$:
   (a) P_0, P_3, P_j for $j \in \{1,2\}$ sample $\mathsf{r}_j \in \mathbb{Z}_{2^\ell}$ while P_0, P_1, P_2 sample $\mathsf{r}_3 \in \mathbb{Z}_{2^\ell}$.
   (b) Locally compute the following:

$$P_0, P_1 : \mathsf{y}_1 = \lambda_{\mathsf{v}_i}^1\mathsf{u}_i^3 + \lambda_{\mathsf{v}_i}^3\mathsf{u}_i^1 + \lambda_{\mathsf{v}_i}^1\mathsf{u}_i^1 - \mu_i^1 + (\mathsf{r}_3 - \mathsf{r}_1)$$
$$P_0, P_2 : \mathsf{y}_2 = \lambda_{\mathsf{v}_i}^2\mathsf{u}_i^3 + \lambda_{\mathsf{v}_i}^3\mathsf{u}_i^2 + \lambda_{\mathsf{v}_i}^3\mathsf{u}_i^3 - \mu_i^3 + (\mathsf{r}_2 - \mathsf{r}_3)$$
$$P_3 : \mathsf{y}_3 = \lambda_{\mathsf{v}_i}^1\mathsf{u}_i^2 + \lambda_{\mathsf{v}_i}^2\mathsf{u}_i^1 + \lambda_{\mathsf{v}_i}^2\mathsf{u}_i^2 - \mu_i^2 + (\mathsf{r}_1 - \mathsf{r}_2)$$

   (c) P_0, P_1 jsnd y_1 to P_3, while P_0, P_2 jsnd $\mathsf{H}(\mathsf{y}_2)$ to P_3.
   (d) P_3 sets flag $=$ `continue` if $\mathsf{H}(\mathsf{y}_2) = \mathsf{H}(-\mathsf{y}_1 - \mathsf{y}_3)$, else flag $=$ `abort` and sends flag to P_0, P_1, P_2. Parties mutually exchange flag and accept the majority value.
   (e) For robust case, if flag $=$ `abort`, then $\mathsf{TTP} = P_1$ (or P_2).

Online:

1. Parties locally compute the following:

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

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 $\llbracket \cdot \rrbracket$-shares.
4. Compute $\llbracket \mathsf{z} \rrbracket = \llbracket \mathsf{z}^1 \rrbracket + \llbracket \mathsf{z}^2 \rrbracket + \llbracket \mathsf{z}^3 \rrbracket$.

Figure 17: Piecewise polynomial evaluation protocol

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.

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

(ii) Sig: We use the MPC-friendly variant of the Sigmoid function [14, 41, 43] which is given below:

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

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 $\mathsf{z} = \mathsf{x}_\mathsf{b}$. The protocol is similar in spirit to Oblivious Transfer primitive. Note that z can be written as $\mathsf{z} = \mathsf{b}(\mathsf{x}_1 - \mathsf{x}_0) + \mathsf{x}_0$. To compute $[\![\cdot]\!]$-sharing of $\mathsf{b}(\mathsf{x}_1 - \mathsf{x}_0)$, parties use an instance of piecewise polynomial protocol ($\Pi_{\mathsf{piecewise}}$, Fig. 17) with $m = 1$. The $[\![\cdot]\!]$-share of z can then be obtained by adding the output of $\Pi_{\mathsf{piecewise}}$ with $[\![\mathsf{x}_0]\!]$.

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

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 $\vec{\mathbf{x}}_2 = (\mathsf{x}_{\lfloor m/2 \rfloor+1},\ldots,\mathsf{x}_m)$.
   - $([\![\mathsf{d}_1]\!]^{\mathbf{B}},\ldots,[\![\mathsf{d}_{\lfloor m/2 \rfloor}]\!]^{\mathbf{B}},[\![\mathsf{y}_1]\!]) = \Pi_{\mathsf{argmin}}([\![\vec{\mathbf{x}}_1]\!])$.
   - $([\![\mathsf{d}_{\lfloor m/2 \rfloor+1}]\!]^{\mathbf{B}},\ldots,[\![\mathsf{d}_m]\!]^{\mathbf{B}},[\![\mathsf{y}_2]\!]) = \Pi_{\mathsf{argmin}}([\![\vec{\mathbf{x}}_2]\!])$.
   - $[\![\mathsf{d}]\!]^{\mathbf{B}} = \Pi_{\mathsf{bitext}}([\![\mathsf{y}_1]\!],[\![\mathsf{y}_2]\!])$.
   - $[\![\mathsf{y}]\!] = \Pi_{\mathsf{obv}}([\![\mathsf{y}_2]\!],[\![\mathsf{y}_1]\!],[\![\mathsf{d}]\!]^{\mathbf{B}})$.
   - $[\![\mathsf{b}_j]\!]^{\mathbf{B}} = \Pi_{\mathsf{Mult}}([\![\mathsf{d}]\!]^{\mathbf{B}},[\![\mathsf{d}_j]\!]^{\mathbf{B}})$; $j \in \{1,\ldots,\lfloor m/2 \rfloor\}$.
   - $[\![\mathsf{b}_j]\!]^{\mathbf{B}} = \Pi_{\mathsf{Mult}}(1 \oplus [\![\mathsf{d}]\!]^{\mathbf{B}},[\![\mathsf{d}_j]\!]^{\mathbf{B}})$; $j \in \{\lfloor m/2 \rfloor+1,\ldots,m\}$.
   - Return $([\![\mathsf{b}_1]\!]^{\mathbf{B}},\ldots,[\![\mathsf{b}_m]\!]^{\mathbf{B}},[\![\mathsf{y}]\!])$.

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*.

D Garbled World

D.1 Garbling scheme and properties

As per Yao's garbling circuit paradigm [57], 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. $\mathsf{Gb}(1^\kappa, \mathsf{Ckt}) \rightarrow (\mathsf{GC}, e, d)$: 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. $\mathsf{En}(x, e) \rightarrow \mathbf{X}$: 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. $\mathsf{Ev}(\mathsf{GC}, \mathbf{X}) \rightarrow \mathbf{Y}$: Ev evaluates the garbled circuit GC on the encoded input $\mathbf{X}$ and produces the encoded output $\mathbf{Y}$.
4. $\mathsf{De}(\mathbf{Y}, d) \rightarrow y$: 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 [8] 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 $C : \{0,1\}^n \rightarrow \{0,1\}^m$ and inputs $x \in \{0,1\}^n$, the following holds.

$$\mathsf{Pr}[\mathsf{De}(\mathsf{Ev}(\mathsf{GC}, \mathsf{En}(x, e)), d) \neq C(x) : (\mathsf{GC}, e, d) \leftarrow \mathsf{Gb}(1^\kappa, C)] < \mathsf{negl}(\kappa)$$

2. A garbling scheme $\mathcal{G}$ is said to be *private* if for all $n \leq \mathsf{poly}(\kappa)$, circuit $C : \{0,1\}^n \rightarrow \{0,1\}^m$, 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 $C : \{0,1\}^n \rightarrow \{0,1\}^m$, input $x \in \{0,1\}^n$ and for all PPT adversary $\mathcal{A}$, the following probability is $\mathsf{negl}(\kappa)$.

$$\mathsf{Pr}\begin{pmatrix} \hat{\mathbf{Y}} \neq \mathsf{Ev}(\mathsf{GC}, \mathbf{X}) & \mathbf{X} = \mathsf{En}(x, e), (\mathsf{GC}, e, d) \leftarrow \mathsf{Gb}(\kappa, \mathsf{Ckt}), \\ \wedge \mathsf{De}(\hat{\mathbf{Y}}, d) \neq \bot & : \quad \hat{\mathbf{Y}} \leftarrow \mathcal{A}(\mathsf{GC}, \mathbf{X}) \end{pmatrix}$$

D.2 2GC Variant

We begin with the 2 GC variant. The protocol for generating garbled sharing of a value appears in Fig. 19.

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 $f'(\mathsf{m_x}, \alpha_\mathsf{x}, \lambda^3_\mathsf{x})$ 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 $[\![\mathsf{z}]\!]^{\mathbf{G}}$.

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} garbling 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 19: Generation of $[\![\mathsf{v}]\!]^{\mathbf{G}}$

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 $K \in \{\mathsf{K}^{0,1}_\mathsf{z}, \mathsf{K}^{1,1}_\mathsf{z}\}$ 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. 20.

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

- For an output wire z, let p^j denote the least significant bit of $\mathsf{K}^{0,j}_\mathsf{z}$ and q^j denote the least significant bit of $\mathsf{K}^{\mathsf{z},j}_\mathsf{z}$ for $j \in \{1,2\}$.
- *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 $\mathsf{z} = \mathsf{p}^j \oplus \mathsf{q}^j$.
- *Reconstruction towards $P_g \in \{P_0, P_3\}$*: P_1 sends q^1 and $\mathcal{H} = \mathsf{H}(\mathsf{K}^{\mathsf{z},1}_\mathsf{z})$ 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 $K \in \{\mathsf{K}^{0,1}_\mathsf{z}, \mathsf{K}^{1,1}_\mathsf{z}\}$ whose least significant bit is q^1 and $\mathsf{H}(K) = \mathcal{H}$.

Figure 20: Output computation: reconstruction of z

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. $\alpha_\mathsf{x}, \lambda^3_\mathsf{x}$) 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. $\mathsf{m_x}$). Since evaluation is local, evaluators obtain $[\![\cdot]\!]^{\mathbf{G}}$-sharing of the GC output at the end of 1 round.

Achieving fairness and robustness. To ensure fairness, we require a fair reconstruction protocol which proceeds as follows. As described in §3.2.1, parties first ensure that all parties are alive. If so, they proceed similar to the protocol in Fig. 20, 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 TTP is identified which can take the computation to completion and deliver the output to all.

D.3 1 GC Variant

The input $\mathsf{x} = \mathsf{x}_1 \oplus \mathsf{x}_2$ for this variant consists of two shares, $\mathsf{x}_1 = \mathsf{m_x} \oplus \lambda_{\mathsf{x}}^2$ and $\mathsf{x}_2 = \lambda_{\mathsf{x}}^1 \oplus \lambda_{\mathsf{x}}^3$, where $\mathsf{m_x}, \lambda_{\mathsf{x}}^1, \lambda_{\mathsf{x}}^2, \lambda_{\mathsf{x}}^3$ are as defined in $[\![\mathsf{x}]\!]^{\mathbf{B}}$. To ensure correct key transfer for the value x_2 held by garbler P_0 and evaluator P_1, garblers P_0, P_3 commit to both keys for x_2 towards P_1, while P_0 sends the opening to the key for x_2. Then, P_1 verifies the consistency of the received commitments and the opening, as it possesses x_2. The protocol appears in Fig. 21.

Protocol $\Pi_{\mathsf{Sh}}^{\mathbf{G}}(P_i, P_j, \mathsf{v})$

1. Garblers in Φ_1 generate keys $\mathsf{K}_{\mathsf{v}}^0, \mathsf{K}_{\mathsf{v}}^1$ using free-XOR technique.
2. If $(P_i, P_j) = (P_2, P_3)$: P_i, P_j jsnd $\mathsf{K}_{\mathsf{v}}^{\mathsf{v}}$ to P_1.
3. If $(P_i, P_j) = (P_0, P_1)$:
   - P_0, P_3 compute commitments on $\mathsf{K}_{\mathsf{v}}^0, \mathsf{K}_{\mathsf{v}}^1$, and jsnd the commitment to P_1.
   - P_0 sends the opening of the commitment for $\mathsf{K}_{\mathsf{v}}^{\mathsf{v}}$ to P_1.
   - P_1 verifies if the received opening information correctly decommits the commitment on $\mathsf{K}_{\mathsf{v}}^{\mathsf{v}}$, where v is held by P_1. Else it `aborts`.
4. Party $P_s \in \Phi_1$ sets $[\![\mathsf{v}]\!]_s^{\mathbf{G}} = \mathsf{K}_{\mathsf{v}}^0$, while P_1 sets $[\![\mathsf{v}]\!]_1^{\mathbf{G}} = \mathsf{K}_{\mathsf{v}}^{\mathsf{v}}$.

Figure 21: Generation of $[\![\mathsf{v}]\!]^{\mathbf{G}}$

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.

Achieving fairness. To ensure fair reconstruction, as in §3.2.1, 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 $\mathsf{K}_{\mathsf{z}}^{\mathsf{z}}$) and $\mathcal{H} = \mathsf{H}(\mathsf{K}_{\mathsf{z}}^{\mathsf{z}})$ 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 $K \in \{\mathsf{K}_{\mathsf{z}}^0, \mathsf{K}_{\mathsf{z}}^1\}$ 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.

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 TTP $= P_2$ 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 TTP.

E Mixed Framework

Table 8 compares the our sharing conversions with Trident. For uniformity, we consider a function, F, to be computed on an ℓ-bit input x using a garbled circuit (GC) in the mixed framework, which gives an ℓ-bit output $\mathsf{y} = \mathsf{F}(\mathsf{x})$, where ℓ denotes the ring size in bits. Let C^{F} denote the corresponding GC. In the table, C^{S2} denotes a 2-input garbled subtraction circuit; $\mathsf{C}^{\mathsf{S2+}}$ denotes 2-input garbled subtraction circuit with its decoding information; C^{S3} denotes 3-input garbled subtraction circuit (with input: $\mathsf{x}, \mathsf{y}, \mathsf{z}$, output: $\mathsf{x} - \mathsf{y} - \mathsf{z}$); C^{A3} denotes 3-input garbled addition circuit; $\mathsf{C}^{1,\dots,i}$ denotes the set of GCs

$\mathsf{C}^1, \ldots, \mathsf{C}^i$; $|\mathsf{C}^{1,\ldots,i}|$ denotes the size of $\mathsf{C}^{1,\ldots,i}$. Note that the cost for Tetrad-R$^{\mathsf{I}}$ is the same as that of Tetrad for conversions not involving the GC. Hence, we omit its details.

Protocol	Reference	Comm. (Preprocessing)	Rounds (Online)	Comm. (Online)
Arithmetic to Garbled to Arithmetic	Trident	$\lvert\mathsf{C}^{\mathsf{S2,S2+,F}}\rvert + 2\ell\kappa + \ell$	2	$\ell\kappa + 3\ell$
	Tetrad$_\mathsf{T}$	$2\lvert\mathsf{C}^{\mathsf{F}}\rvert + 6\ell\kappa + \ell$	1	$2\ell\kappa + \ell$
	Tetrad$_\mathsf{C}$	$\lvert\mathsf{C}^{\mathsf{F}}\rvert + 3\ell\kappa + \ell$	2	$\ell\kappa + 2\ell$
Arithmetic to Garbled to Boolean	Trident	$\lvert\mathsf{C}^{\mathsf{S2,F}}\rvert + 2\ell\kappa + \ell$	2	$\ell\kappa + 3\ell$
	Tetrad$_\mathsf{T}$	$2\lvert\mathsf{C}^{\mathsf{F}}\rvert + 6\ell\kappa + \ell$	1	$2\ell\kappa + \ell$
	Tetrad$_\mathsf{C}$	$\lvert\mathsf{C}^{\mathsf{F}}\rvert + 3\ell\kappa + \ell$	2	$\ell\kappa + 2\ell$
Boolean to Garbled to Arithmetic	Trident	$\lvert\mathsf{C}^{\mathsf{S2+,F}}\rvert + 2\ell\kappa + \ell$	2	$\ell\kappa + 3\ell$
	Tetrad$_\mathsf{T}$	$2\lvert\mathsf{C}^{\mathsf{F}}\rvert + 6\ell\kappa + \ell$	1	$2\ell\kappa + \ell$
	Tetrad$_\mathsf{C}$	$\lvert\mathsf{C}^{\mathsf{F}}\rvert + 3\ell\kappa + \ell$	2	$\ell\kappa + 2\ell$
Boolean to Garbled to Boolean	Trident	$\lvert\mathsf{C}^{\mathsf{F}}\rvert + 2\ell\kappa + \ell$	2	$\ell\kappa + 3\ell$
	Tetrad$_\mathsf{T}$	$2\lvert\mathsf{C}^{\mathsf{F}}\rvert + 6\ell\kappa + \ell$	1	$2\ell\kappa + \ell$
	Tetrad$_\mathsf{C}$	$\lvert\mathsf{C}^{\mathsf{F}}\rvert + 3\ell\kappa + \ell$	2	$\ell\kappa + 2\ell$
Arithmetic to Boolean	Trident	$3\ell\log_2\ell + 2\ell$	$1 + \log_2\ell$	$3\ell\log_2\ell + \ell$
	Tetrad	$\mathsf{u}_1{}^* + \ell$	$\log_4\ell$	$3\mathsf{u}_3{}^* + \ell$
	Tetrad-R$^{\mathsf{II}}$	$\mathsf{u}_2{}^* + \ell$	$\log_4\ell$	$3\mathsf{u}_3{}^* + \ell$
Boolean to Arithmetic	Trident	$3\ell^2 + \ell$	1	3ℓ
	Tetrad	$3\ell^2 + \ell$	1	3ℓ
	Tetrad-R$^{\mathsf{II}}$	$3\ell^2 + \ell$	1	3ℓ

– Notations: ℓ - size of ring in bits, κ - computational security parameter.
*: $\mathsf{u}_1 = 2\mathsf{n}_2 + 9\mathsf{n}_3 + 24\mathsf{n}_4$, $\mathsf{u}_2 = 3\mathsf{n}_2 + 12\mathsf{n}_3 + 33\mathsf{n}_4$, $\mathsf{u}_3 = \mathsf{n}_2 + \mathsf{n}_3 + \mathsf{n}_4$, where $\mathsf{n}_2 = 216, \mathsf{n}_3 = 184, \mathsf{n}_4 = 179$ denote the number of AND gates in the optimized adder circuit [45] with 2, 3, 4 inputs, respectively.

Table 8: Sharing conversions of Trident and Tetrad.

Arithmetic to Boolean Conversion. The protocol for arithmetic to boolean conversion appears in Fig. 22.

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 22: Arithmetic to Boolean Conversion

Boolean to Arithmetic Conversion. The protocol for arithmetic to boolean conversion appears in Fig. 23.

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 $[\![\mathsf{y}]\!] = [\![\mathsf{y}^1]\!] + [\![\mathsf{y}^2]\!] + [\![\mathsf{y}^3]\!]$.

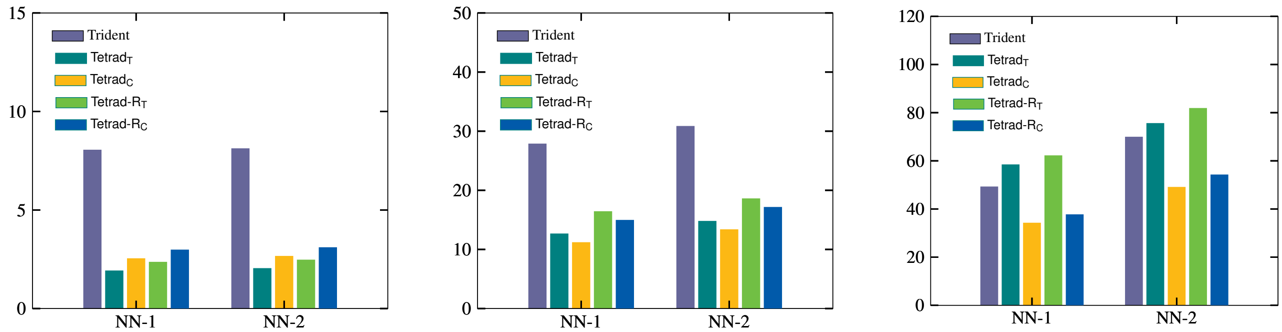
Figure 23: Boolean to Arithmetic Conversion

We remark that the protocol Π_{B2A} can be used to efficiently generate edaBits [21] 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.

F Additional Benchmarking

Training and Inference of NN. An NN can be divided into various layers, where each layer contains a predefined number of nodes. These nodes are a linear function composed of a non-linear "activation" function. The nodes at the input layer or the first 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 [43]. To perform the division, we switch from arithmetic to garbled world and then use a division garbled circuit [48] 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 [15, 41, 43, 46, 56] for formal details.



(a) Online Execution Time ($\mathsf{PT}_{\mathsf{on}}$) (b) Cumulative Time ($\mathsf{CT}_{\mathsf{tot}}$) (c) Monetary Cost (Cost)

Figure 24: Training of NN-1 and NN-2: in terms of $\mathsf{PT}_{\mathsf{on}}$, $\mathsf{CT}_{\mathsf{tot}}$, and Cost (cf. Table 4)

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.

Benchmarking of NN-1 Training. Table 9 shows the online throughput of neural network (NN-1) training over varying batch sizes and feature sizes using synthetic datasets. We find that both Tetrad_T, Tetrad_C are up to $1.8\times$ higher in throughput. 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 throughput over Trident decreases a bit.

Batch Size	Features	Trident	Tetrad_T	Tetrad_C	Tetrad-R_T	Tetrad-R_C
128	10	1905.58	5407.35	5271.88	5407.35	5138.07
	100	1905.58	5152.29	5029.14	5152.28	5029.14
	1000	1904.4	3500.89	3443.6	3500.89	3443.6
256	10	1905.58	2818.4	2744.87	2818.4	2744.87
	100	1905.58	2747.5	2677.58	2747.5	2677.58
	1000	1849.78	2195.3	2150.43	2195.3	2150.43

Table 9: Online throughput (TP) of NN-1 training (iterations per minute) over various batch sizes (128, 256) and feature sizes (10,100,1000).

Benchmarking of Comparison operations. Table 10 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.

Interestingly, beyond a depth of roughly 100, Tetrad_T, Tetrad-R_T start performing in every metric, especially monetary cost, over Tetrad_C, Tetrad-R_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 10 is that if one were to use a DNN with a depth of over 100, Tetrad_T, Tetrad-R_T become the optimal choices.

Depth	Parameter	Trident	Tetrad_T	Tetrad_C	Tetrad-R_T	Tetrad-R_C
128	$\mathsf{PT}_{\mathsf{on}}$	3.55	0.53	0.93	0.53	0.93
	$\mathsf{CT}_{\mathsf{tot}}$	9.6	1.06	1.85	1.06	1.85
	Cost	0.49	0.05	0.09	0.05	0.09
1024	$\mathsf{PT}_{\mathsf{on}}$	28.42	4.23	7.41	4.23	7.41
	$\mathsf{CT}_{\mathsf{tot}}$	76.79	8.47	14.82	8.47	14.82
	Cost	3.89	0.43	0.75	0.45	0.76
8192	$\mathsf{PT}_{\mathsf{on}}$	227.34	33.87	59.27	33.87	59.27
	$\mathsf{CT}_{\mathsf{tot}}$	614.3	67.76	118.56	67.76	118.56
	Cost	31.27	3.48	6.03	3.49	6.03

Table 10: Benchmarking of comparisons over various depths. Each depth has 128 comparisons. Time is reported in minutes, and monetary cost in USD.

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. 6), $\mathcal{F}_{\mathsf{jsnd}}$ (Fig. 26) 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 $\mathcal{F}_{\mathsf{GOD}}$ 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 TTP is identified during the simulation, the simulator stops and returns the function output to the adversary on behalf of the TTP as per $\mathcal{F}_{\mathsf{jsnd}}$.

Ideal jsnd **Functionality.** The ideal jsnd functionality for fairness security appears in Fig. 25 and that for the robust setting appears in Fig. 26.

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

$\mathcal{F}_{\mathsf{jsnd}}$ interacts with the servers 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 server 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 25: Ideal functionality for jsnd in Tetrad

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

$\mathcal{F}_{\mathsf{jsnd}}$ interacts with the servers 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 server P_l, while it receives $(\mathsf{select}, \mathsf{ttp})$ from $\mathcal{S}$. Here ttp is a boolean value, with a 1 indicating that $\mathsf{TTP} = P_l$ 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 26: Ideal functionality for robust jsnd [33]

G.1 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.

Sharing Protocol (Π_{Sh}, Fig. 10). 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 $\mathcal{F}_{\mathsf{GOD}}$ on input $(\mathsf{Input}, \mathsf{v})$ 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 27: 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 $\mathcal{F}_{\mathsf{GOD}}$ on input $(\mathsf{Input}, \mathsf{v})$ 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 28: 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.

Joint Sharing Protocol: 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 $\mathcal{F}_{\mathsf{jsnd}}$.

Multiplication Protocol ($\Pi^{\mathsf{R}}_{\mathsf{Mult}}$, Fig. 12).

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.

– Emulates two instances of $\mathcal{F}_{\mathsf{jsnd}}$ with $\mathcal{A}$ as one sender to send $\mathsf{w}_1, \mathsf{w}_2$ to P_3.

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 29: 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.

– Emulates one instance of $\mathcal{F}_{\mathsf{jsnd}}$ with $\mathcal{A}$ as one sender to send w_1 to P_3.

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 30: 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.

– Emulates two instances of $\mathcal{F}_{\mathsf{jsnd}}$ with $\mathcal{A}$ as receiver to send $\mathsf{w}_1, \mathsf{w}_2$ to $\mathcal{A}$.

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

Reconstruction Protocol (Π_{Rec}, Fig. 11). Using the input of $\mathcal{A}$ obtained during simulation of sharing protocol, $\mathcal{S}_{\Pi_{\mathsf{Rec}}}$ invokes $\mathcal{F}_{\mathrm{GOD}}$ 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.

G.2 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 $\mathcal{F}_{\mathrm{GOD}}$ 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:

1. Generation of boolean shares for the input.
2. Transfer of keys and GC to the evaluator.
3. 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.

Generation of boolean shares for the input. This simulation proceeds as per the simulation of the boolean world mentioned in §G.1.

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 32: 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 $\mathcal{F}_{\mathrm{GOD}}$ 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.

– $\mathcal{S}^{P_1}_{\mathsf{Ev}}$ 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.

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

Figure 33: Simulator $\mathcal{S}^{P_1}_{\mathsf{Ev}}$ for corrupt P_1

Output computation.

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

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

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

Figure 34: Simulator $\mathcal{S}^{P_0}_{\mathsf{Rec}}$ 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 $\mathcal{F}_{\mathrm{GOD}}$.

Figure 35: Simulator $\mathcal{S}_{\mathsf{Rec}}^{P_1}$ for corrupt P_1

Indistinguishability argument. We argue that $\textsc{ideal}_{\mathcal{F},\mathcal{S}_\Pi} \overset{c}{\approx} \textsc{real}_{\Pi,\mathcal{A}}$ when $\mathcal{A}$ corrupts P_1 based on the following series of intermediate hybrids.

$\textsc{hyb}_0$: Same as $\textsc{real}_{\Pi,\mathcal{A}}$.

$\textsc{hyb}_1$: Same as $\textsc{hyb}_0$, except that P_0, P_2, P_3 use uniform randomness instead of pseudo-randomness to sample values not known to P_1.

$\textsc{hyb}_2$: Same as $\textsc{hyb}_1$ 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 $\textsc{hyb}_2 := \textsc{ideal}_{\mathcal{F},\mathcal{S}_\Pi}$, to conclude the proof we show that every two consecutive hybrids are indistinguishable.

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

$\textsc{hyb}_1 \overset{c}{\approx} \textsc{hyb}_2$: The difference between the hybrids is in the way $(\mathsf{GC}_1, \mathbf{X}_1, d_1)$ is generated. In $\textsc{hyb}_1$, $(\mathsf{GC}_1, e_1, d_1) \leftarrow \mathsf{Gb}(1^\kappa, \mathsf{Ckt})$ is run. In $\textsc{hyb}_2$, 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 $\textsc{ideal}_{\mathcal{F},\mathcal{S}_\Pi} \overset{c}{\approx} \textsc{real}_{\Pi,\mathcal{A}}$ when $\mathcal{A}$ corrupts P_0 based on the following series of intermediate hybrids.

$\textsc{hyb}_0$: Same as $\textsc{real}_{\Pi,\mathcal{A}}$.

$\textsc{hyb}_1$: Same as $\textsc{hyb}_0$, except that P_1, P_2, P_3 use uniform randomness instead of pseudo-randomness to sample values not known to P_0.

$\textsc{hyb}_2$: Same as $\textsc{hyb}_1$ 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 $\textsc{hyb}_2 := \textsc{ideal}_{\mathcal{F},\mathcal{S}_\Pi}$, to conclude the proof we show that every two consecutive hybrids are indistinguishable.

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

$\textsc{hyb}_1 \overset{c}{\approx} \textsc{hyb}_2$: The difference between the hybrids is that in $\textsc{hyb}_1$, 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 $\textsc{hyb}_2$, 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.